Algorithmen, Datenstrukturen und Objekte

Programmieren in Python

Gregor Kuhlmann

1 Python – eine Einführung

Python ist sehr moderne Sprache. Die Entwicklung begann 1999. Es gibt zwei Entwicklungslinien, die mit Version 2 bzw. Version 3 bezeichnet werden. Die Version 3.0 wurde 2008 veröffentlicht und wird ständig weiterentwickelt. Ergänzt wird die Sprache durch umfangreiche Bibliotheken wie Numpy oder Matplotlib.

Der Name Python nichts mit der Schlange zu tun. Der Entwickler Guido von Rossum war ein Fan der Monty Python Show. Sein Ziel war es, eine Sprache zu entwickeln, die mächtig, aber auch leicht zu erlernen ist.

Python ist eine Programmiersprache, mit der sowohl funktions- als auch objektorientiert programmiert werden kann. Der Programmcode wird auch Skript genannt. Python arbeitet nicht mit einem Compiler, sondern mit einem Interpreter.

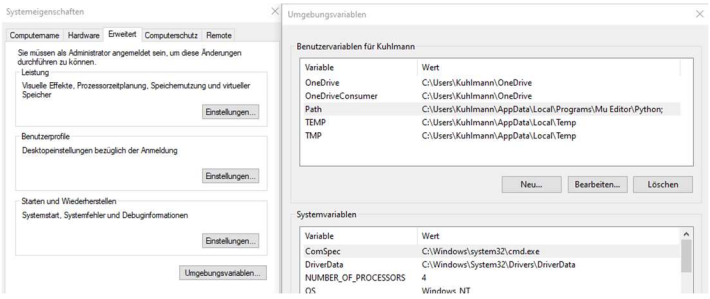
Python ist weit verbreitet im Einsatz. Es läuft auf den drei Betriebssystemen Windows, Linux und OS-X sowie auf Handys und Tablets

2 Entwicklungsumgebung

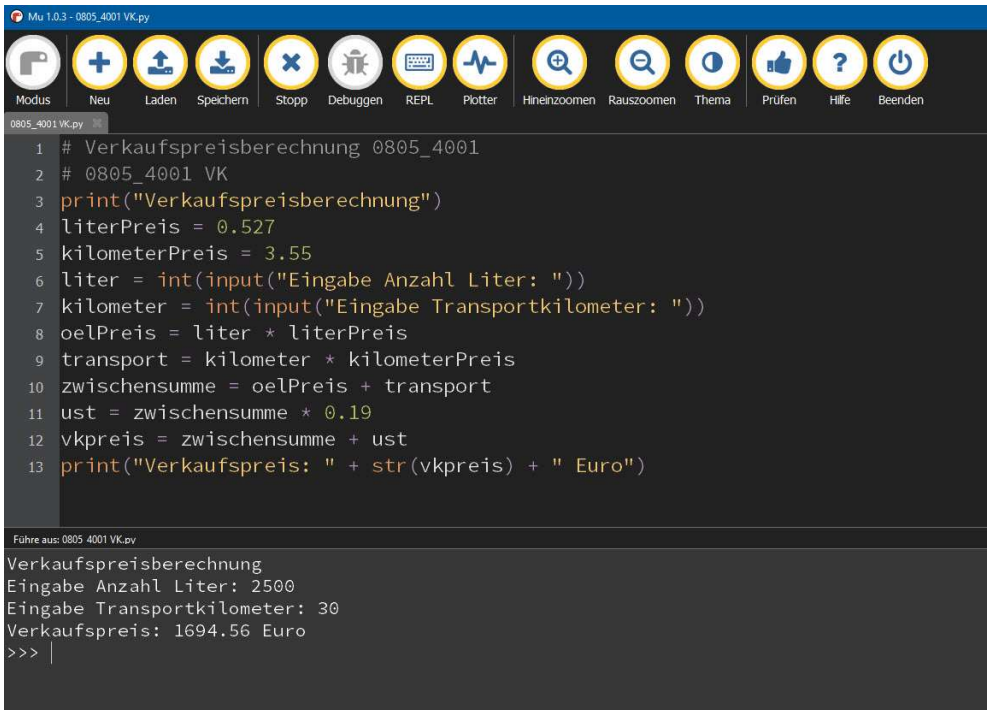
2.1 Installation

Bei Windows-Rechner muss Python installiert werden. Bei den anderen Betriebssystemen gehört es häufig mit zum Lieferumfang. Die Installation ist in der Regel unproblematisch, nachdem die Sicherheitsbedenken von Windows überwunden sind.

Wird die Entwicklungsumgebung Mu verwendet, wird bei ihrer Installation auch Python installiert. Möglicherweise muss die Umgebungsvariable angepasst werden, wenn mu nicht korrekt startet.



2.1.1 Entwicklungsumgebung Mu

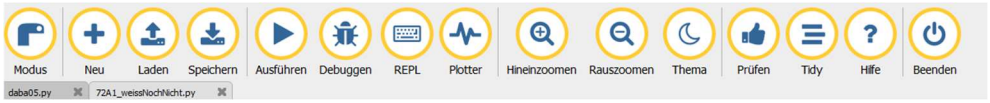


Die Entwicklungsumgebung (IDE = Integrated Development Environment) weist zwei Bereiche auf. Unterhalb der Menüleiste befindet sich der Editierbereich. Hier geben Sie den Programmcode bzw. das Skript ein.

Unterhalb des Editierbereichs befindet sich die sogenannte Shell bzw. Python-Konsole. Hier erfolgt die Ein- und Ausgabe von Daten.

In diesen Bereich können Sie Python-Anweisungen schreiben, die direkt vom Interpreter ausgewertet werden. Man spricht dann auch vom interaktiven Modus oder Direktmodus.

2.1.2 Editier-Modus



Mu kennt verschiedene Arbeitsarten. Sie stellen diese ein über den Schaltknopf Modus. Hier wählen Sie *Python 3.*

Zur Eingabe eines neuen Programms klicken Sie auf die Schaltfläche *Neu*. Nach Abschluss der Eingabe können Sie durch einen Klick auf die Schaltfläche *Ausführen* das Programm ausführen. Zuvor fordert Mu Sie auf, das

Programm zu speichern. Geben Sie einen passenden Dateinamen an. Die Speicherung erfolgt im Unterverzeichnis mu_code. Anschließend führt der Interpreter das Programm aus.

Findet der Interpreter einen Fehler, zeigt er ihn in dem Direktbereich an. Die Korrektur nehmen Sie dann im Editierbereich vor. Zur erneuten Ausführung klicken Sie auf die Schaltfläche *Stopp* und anschließend auf *Ausführen*.

Die Prüfung des Codes können Sie auch durch Anklicken der Schaltfläche *Prüfen* starten.

Gestaltung des Editierbereiches

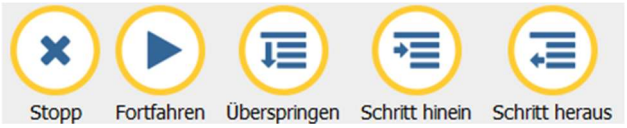
Mittels der Schaltflächen *Hineinzoomen*, *Rauszoomen* und *Thema* können Sie die Schriftgröße sowie die Schriftfarbe verändern.

Die Schaltfläche Tidy hilft Ihnen, den Code stilistisch entsprechend den Layout-Empfehlungen von Python darzustellen.

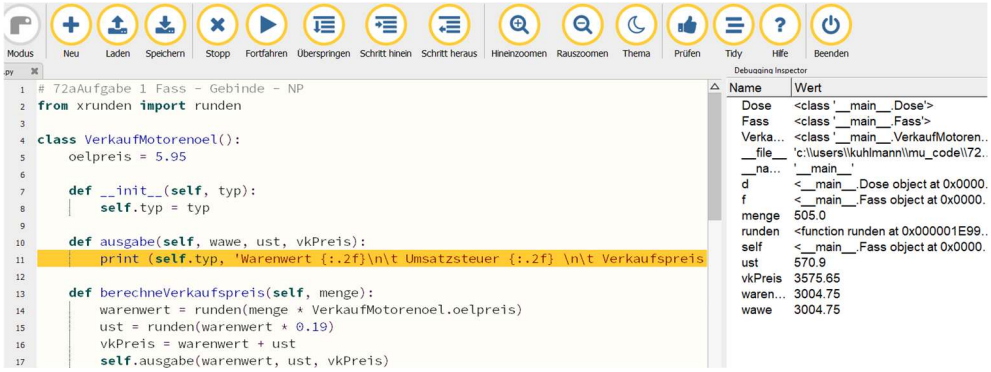
2.1.3 Debugging

Es gibt zwei Arten von Fehlern – Syntaxfehler und Logikfehler. Syntaxfehler findet der Interpreter. Liegt ein Fehler in der Programmlogik vor, so dass kein oder ein falsches Ergebnis angezeigt wird, müssen Sie den Programmablauf prüfen. Es kann zum Beispiel eine Auswahlbedingung falsch formuliert oder eine Schleifenbedingung inkorrekt sein. Um derartige Fehler zu finden, ist der Debugger hilfreich.

Im Debugger-Modus verändert sich die Menüleiste.



Mittels der Schaltfläche *Schritt hinein* durchlaufen Sie die einzelnen Programmzeilen. Im rechten Teil des Editierbereichs wird in der Spalte *debugging inspector* der Ablauf der Programmbearbeitung und der Inhalt der Variablen angezeigt.



Soll der Ablauf in einem Modul oder in einer Funktion nicht angezeigt werden, klicken Sie auf *Überspringen*. Um aus einem Modul herauszugehen, klicken Sie auf *Schritt heraus*. Der Klick auf *Fortfahren* beendet das Debugging.

2.2 Entwicklungsumgebung PyCharm

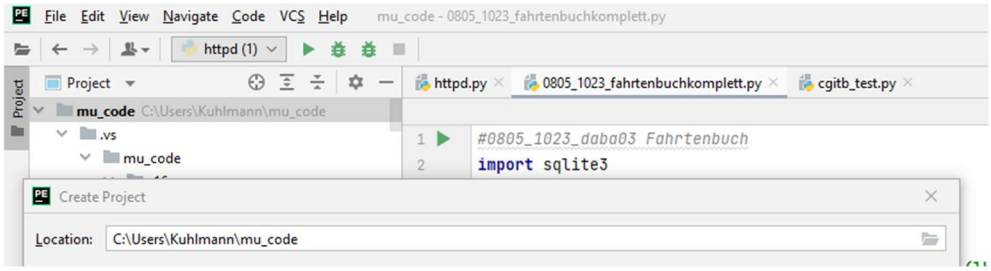
PyCharm ist in drei Versionen verfügbar. Für professionelle Entwickler wird eine kostenpflichtige Version angeboten. Die Community und die Edu-Edition sind Open-Source-Projekte.

Möglicherweise muss nach der Installation von PyCharm in der Umgebungs-/Benutzervariablen angegeben werden, in welchem Verzeichnis sich das Python-System befindet.

PyCharm ist ein vielseitig einsetzbares Programm. Deshalb wird von der Standardeinstellung ausgegangen und gezeigt, wie Python-Programme editiert und ausgeführt werden.

2.2.1 Eingabe eines Programms

PyCharm arbeitet projekt-orientiert. In dem Projektordner werden alle Dateien gespeichert. Wurde noch kein Projekt angelegt, rufen Sie im Menü *File* den Unterpunkt *New Project* auf.

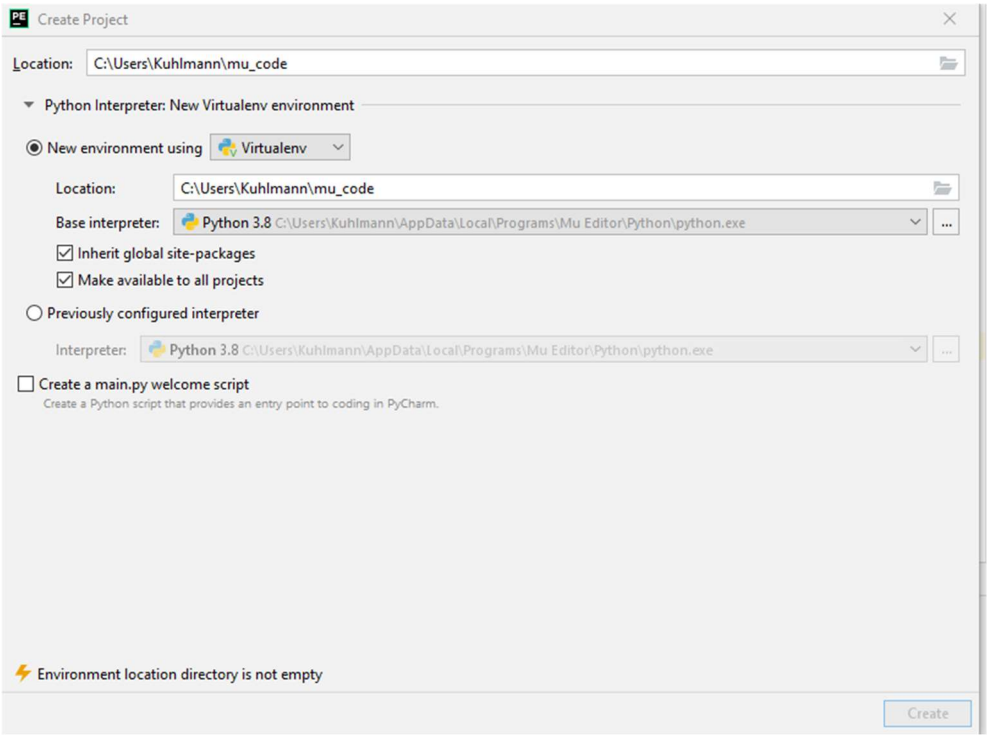


In dem Fenster *Create Project* legen Sie das Verzeichnis fest, in dem die zu erstellenden Programme gespeichert werden sollen. Die übrigen

Einstellungen lassen Sie unverändert. Falls kein Interpreter angezeigt wird, müssen Sie den Pfad zu python.exe angeben.

Die Option *Create a main.py* können Sie ausschalten.

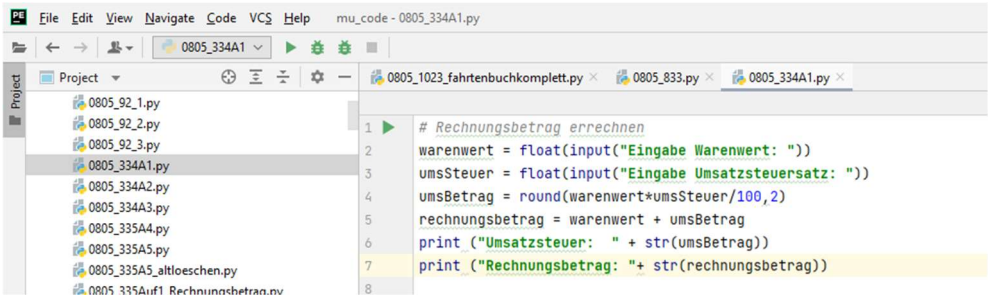
Abschluss mit Create. In einem pop-up-Menü können Sie auswählen, ob ein neues Fenster aufgemacht werden soll.



Die Option *Create a main.py* können Sie ausschalten.

Abschluss mit Klick auf Create. In einem pop-up-Menü können Sie auswählen, ob ein neues Fenster aufgemacht werden soll.

In der zweiten Menüzeile des Hauptmenüs steht nun der Projektname. Mit einem Rechtsklick auf den Namen öffnen Sie ein Auswahlmenü, in dem Sie *New/Python File* und geben Sie in dem Dialogfenster *New Python File* den Namen der Programmdatei ein. Die Datei wird angelegt und Sie können mit der Eingabe beginnen.

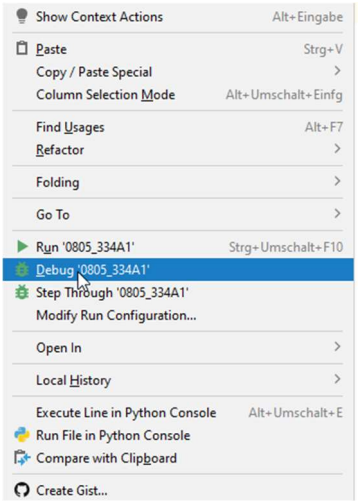


Nach Abschluss der Eingaben rufen Sie mit einem Rechtsklick ein Befehlsmenü auf, in dem Sie den Befehl *run* klicken.

Das Ergebnis wird auf der Konsole angezeigt.

Das eingegebene Programm wird automatisch gespeichert. Um ein gespeichertes Programm laden, klicken Sie im Hauptmenu auf *File*. Im Befehlsmenü klicken Sie auf *Open*. Sie können natürlich auch in dem Objektbrowser das gewünschte Programm doppel-klicken und es in das Editier-Fenster laden.

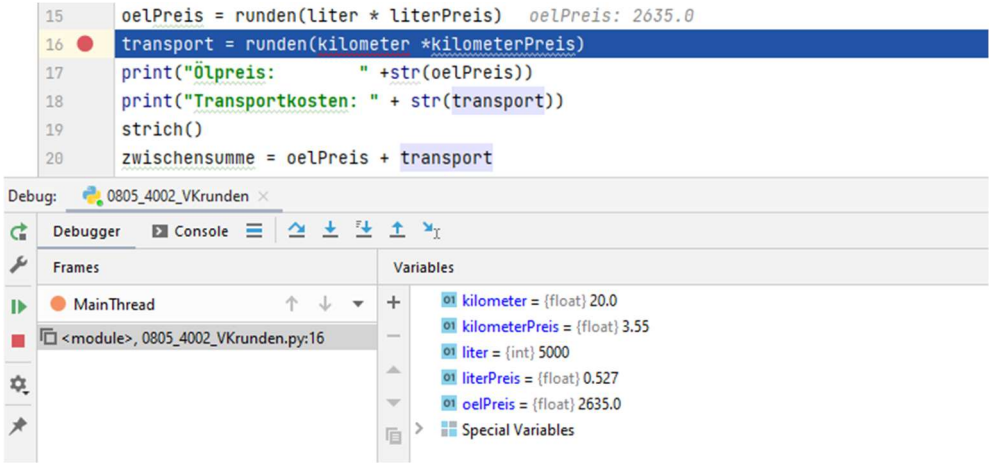
2.2.2 Debugging



Für die Fehlersuche stellt PyCharm einen Debugger zur Verf ügung, den Sie aus dem Befehlsmenü aufrufen.

Sie können eine Anweisung als Startpunkt für die schrittweise Bearbeitung festlegen. Dazu klicken Sie in die Vorspalte.

In der Konsole wird der Zustand des Programms sowie der Inhalt der Variablen angezeigt. In der Debugger-Zeile werden Ihnen Pfeiltasten angezeigt, mit denen Sie den Programmablauf steuern können.



Bei der Education-Version können Sie auch das Programm Schritt für Schritt ausführen und dabei beobachten, wie sich die Werte in den Variablen verändern

2.3 Auf der Konsole arbeiten

Direktmodus

Beide hier besprochenen Entwicklungsumgebungen Mu und PyCharm verfügen über eine Konsole, auf der im Direktmodus gearbeitet werden kann.

Beim Direktmodus werden die Anweisungen direkt ausgeführt.

Beispiele

Fall 1

```
>>> a = 5
>>> b = 6
>>> c = a +b
>>> print (c)
11
```

Fall 2

```
>>> a=6
>>> b=9
>>> if a>b:
...   print(b-a)
... else:
...   print(a-b)
...
-3
```

Fall 3

```
>>> for i in ["ab","hg","cd","ef"]:
...     print(i)
...
ab
hg
cd
ef
```

Wie aus dem obigen Beispiel zu ersehen ist, besteht ein Programm aus Anweisungen. Diese werden hinter den drei Winkeln >>> eingetragen.

Die drei Beispiele zeigen, dass sich der Direktmodus gut für kurze Programmierübungen eignet.

Fall 1

Im ersten Fall werden zwei Zahlen vereinbart. Das System stellt einen Speicherplatz für die 5 zur Verfügung, der mit dem Namen a bezeichnet wird. Mit Hilfe des Bezeichners kann per Programm auf den Speicherplatz zugegriffen werden. Der Bezeichner wird auch Variable genannt. Sie verweist auf einen Speicherplatz, dessen Inhalt durch eine Zuweisung verändert wird.

Das Gleichheitszeichen ist ein **Zuweisungsoperator**.

In der dritten Zeile werden die Inhalte der Variablen addiert und der Variablen c zugewiesen. Schließlich wird der Inhalt der Variablen c ausgegeben.

Die Namen der Variablen sind frei wählbar. Lediglich Schlüsselwörter der Programmiersprache dürfen nicht als Variablennamen verwendet werden.

Fall 2

Im zweiten Fall wird eine Auswahlanweisung dargestellt. Wenn der Wert in der Variablen a größer ist als der in der Variablen b, dann soll a von b subtrahiert werden. Anderenfalls wird b von a subtrahiert.

Fall 3

Im dritten Fall werden die Elemente einer Liste ausgegeben. Die Liste besteht aus 5 Zeichenfolgen. Der Datentyp Zeichen wird durch Anführungszeichen oder Hochkommata gekennzeichnet, beispielweise “a“, ‘B‘. Mehrere Zeichen in eine Folge werden Zeichenkette oder auch **String** genannt. “ab“ oder auch “Berufskolleg“ sind Beispiele für Zeichenketten.

Die Befehlseingaben in der Shell bzw. Konsole werden nicht gespeichert. Wollen Sie den Programmcode speichern, ist er im Editor-Fenster einzugeben.

3 Lineare Programme

3.1 Anweisungen zur Ein- und Ausgabe von Daten

3.1.1 Eingabe

Der Befehl zur Eingabe von Daten lautet input(). In der Eingabeanweisung die Variable anzugeben, die den Eingabewert aufnimmt. Die Eingabe wird also wie eine Wertzuweisung behandelt.

Fall 1

```
a = input()       # 4
b = input()       # 5
print(a, b)
----------------------
Ausgabe: 4    5
```

Fall 2

```
a = input("Eingabe einer Zahl a: ")
b = input("Eingabe einer Zahl b: ")
print(a+b)
----------------------------------------
Ausgabe 45
```

Fall 3

```
a = int(input("Eingabe einer Zahl a: ") )
b = int(input("Eingabe einer Zahl b: "))
print(a+b)
-----------------------------------------
Ausgabe: 9
```

input()

Der Befehl, mit dem einer Variablen ein Wert per Tastatur zugewiesen wird, lautet:

```
variablenname = input()
```

Die Eingabe erfolgt in der Shell bzw. Konsole. Im Fall 1 wird der eingegebene Wert in der Variablen a gespeichert. Der Eingabewert wird als Zeichen gespeichert, auch wenn er eine Ziffer ist.

Im Fall 2 wird der Nutzer darauf hingewiesen, dass er etwas eingeben soll. Runde Klammern umschließen den Hinweis. Der Text steht in Anführungszeichen.

Die eingegebenen Werte werden den Variablen zugewiesen. Hier wurde für a eine 4, für b eine 5 eingegeben.

Im Beispiel 3 wurde zusätzlich eine Typangabe in die Eingabeanweisung eingefügt. Hier wird die Eingabe von Integer-Werten, also Ganzzahlen erwartet.

3.1.2 Ausgabe

print()

Die Ausgabe von Daten erfolgt mittels der print()-Anweisung. Die auszugebenden Daten stehen innerhalb der runden Klammern.

Im Fall 1 werden die Inhalte von zwei Variablen ausgegeben. Die Namen der Variablen sind durch Komma voneinander getrennt.

print(a+b)

Im Fall 2 sollte eine Summe ausgegeben werden. Die Ausgabe ‘ 45‘ überrascht wohl. Die Erklärung ist einfach: Alle per input() eingegebenen Ziffern werden als Zeichen übernommen, auch wenn sie numerische Werte darstellen sollen.

Bei der Ausgabe werden die Zeichen einfach aneinandergehängt. Das wird auch als Konkatenation bezeichnet.

Typbezeichnung int und float

Soll eine Eingabe als Zahl bzw. als numerischer Wert gelten, dann wird eine Typangabe vor den Eingabebefehl gesetzt. Python unterscheidet Ganzzahlen von reellen Zahlen. Die Typangabe für Ganzzahl lautet **int**, für reelle Zahl **float**.

Benennung der Ausgabe

In dem Fall 3 wird nur eine „nackte“ Zahl ausgegeben. Möchten Sie auch ihre Bedeutung in dem Ausgabebefehl erklären, können Sie einen Text dazuschreiben:

```
print(“Ergebnis der Addition von a und b lautet: “ + str(a+b))
```

Der erste Teil der Ausgabe ist eine Folge von Zeichen, eine **Zeichenkette**, der zweite Teil ein numerischer Wert. Diese Mischung von Datentypen kann die print-Anweisung nicht verarbeiten und meldet einen Fehler. Deshalb muss die Zahl in eine Zeichenkette, englisch **String**, umgewandelt werden. Die Typumwandlung besorgt die Funktion **str().**

3.1.3 Zusammenfassung

- Python verlangt keine explizite Variablenvereinbarung. Es ist aber ratsam, den numerischen Eingabewert als int- oder float-Wert zu qualifizieren.
- Daten, die mittels des input()-Befehls eingegeben werden, werden als Zeichen angesehen. Soll die Eingabe als numerischer Wert behandelt werden, muss mittels int() oder float() der numerische Typ angegeben werden.
- Im Ausgabebefehl print() können entweder nur numerische oder alphabetische Zeichen ausgegeben werden. Bei einer gemischten Ausgabe müssen numerische Werte in Zeichenketten (String) umgeformt werden. Die Umformungsfunktion lautet str().

- Python kennt als einfache Datentypen Ganzzahlen (int) oder reelle Zahlen (float) sowie Zeichen und Zeichenkette (String).

3.2 Rechenoperationen

3.2.1 Einfache Rechenoperationen

Das folgende Programm gibt eine Übersicht, wie Rechenoperationen formuliert werden.

```
# Rechenoperationen
a = int(input("Eingabe einer Zahl a: ") )
b = int(input("Eingabe einer Zahl b: "))
summe = a + b
differenz = a - b
produkt = a * b
quotient = a/b
a += b              # Verkürzte Schreibweise    für a = a + b
a -=b               #                           für a = a - b
a *= b              #                           für a = a * b
a /= b              #                           für a = a/b
a = 9.30            # Wertzuweisung einer float-Zahl – Dezimalpunkt!
ganzDiv = a // b    # Ganzzahlige Division
divRest = a % b     # Ermittlung des Divisionsrestes
```

Die erste Zeile des Skripts ist ein **Kommentar**. Er wird durch ein Doppelkreuz gekennzeichnet und sollen das Lesen bzw. das Verständnis eines Skripts erleichtern. Kommentare werden nicht vom Interpreter ausgewertet.

Kommentare, die sich über mehrere Zeilen erstrecken, werden mit drei Anführungszeichen eingeleitet und abgeschlossen.

Anwendung

Beispiel: In der Ölgroßhandlung Klender benötigt man ein Programm, mit dem der Verkaufspreis einer Öllieferung berechnet wird. Der Preis pro Liter Öl beträgt 0,52 ct. Für den Transport werden 3,55 Euro pro Kilometer berechnet.

Einzugeben sind: Anzahl Liter sowie Entfernungskilometer.

Algorithmus

Verkaufspreisberechnung
Konstanten: Literpreis= .527; kilometerpreis = 3.55
Eingabe Liter, Transportkilometer
Berechne Ölpreis
Berechne Transportkosten
Zwischensumme = Ölpreis + Transportkosten
Berechne Umsatzsteuer 19 %
Berechne Verkaufspreis = Zwischensumme + Umsatzsteuer
Ausgabe Verkaufspreis

Programm:

```
1.  #Verkaufspreisberechnung 0805_301
2.  #O805_301 VK
3.  print ("Verkaufspreisberechnung")
4.  literPreis = 0.527
5.  kilometerPreis = 3.55
6.  liter = int(input("Eingabe Anzahl Liter: "))
7.  kilometer = int(input("Eingabe Transportkilometer: "))
8.  oelPreis = liter * literPreis
9.  transport = kilometer *kilometerPreis
10. zwischensumme = oelPreis + transport
11. ust = zwischensumme * 0.19
12. vkpreis = zwischensumme + ust
13. print("Verkaufspreis: " + str(vkpreis) + " Euro")
```

Ausgabe

Erläuterungen

Es empfiehlt sich, im Programmkopf die Aufgabenstellung kurz zu beschreiben, für die das Programm ausgelegt ist. Ferner sollte der Dateiname, unter dem das Programm gespeichert wird, vermerkt werden. Kommentarzeilen werden mit einem Doppelkreuz # eingeleitet.

Die Namen der Variablen werden klein geschrieben. Bei zusammengesetzten Bezeichnungen ist es üblich, die weiteren Namensbestandteile mit Großbuchstaben zu beginnen. Das erhöht die Lesbarkeit.

Beispiel

literPreis, kilometerPreis

Oftmals wird empfohlen, die Namensbestandteile einer Variablen durch Unterstrich zu verbinden, beispielsweise liter_preis. Auf die Lauffähigkeit eines Programms hat die Schreibweise keinen Einfluss, wohl aber für die Lesbarkeit des Skripts.

Die Variablen *literPreis* und *kilometerPreis* beinhalten vom Programm her festgesetzte Werte. Sie können nur durch eine Programmänderung verändert werden. *literPreis* und *kilometerPreis* kann man deshalb auch als Konstanten bezeichnen.

Die beiden Eingabeanweisungen in den Zeilen 6 und 7 enthalten neben Bedienhinweisen auch die Anweisung an den Interpreter, dass die Eingaben als Integer-Werte zu speichern sind.

Im Ausgabebefehl in Zeile 13 muss der Inhalt der Variablen *vkpreis* in eine Zeichenkette umgewandelt werden, da in einer Ausgabeanweisung entweder nur Zeichen oder nur Zahlen ausgegeben werden können.

Fehlerbehandlung

Es ist fast normal, dass nach der Eingabe des Skripts der Programmstart mit Fehlerhinweisen endet.

Bevor das Programm ausgeführt wird, prüft Python, genauer: der Interpreter, das Skript auf syntaktische Richtigkeit. Stellt Python Fehler fest, werden sie auf der Konsole angezeigt.

Beispiel

Wäre die Zeile 6 wie folgt geschrieben

```
6 liter = int(input("Eingabe Anzahl Liter: ")
```

würde angezeigt, dass in Zeile 7 ein Fehler vorliegt. In der Zeile 7 ist aber alles in Ordnung. Der Fehler liegt in der vorhergehenden Zeile. Der Interpreter kann aber erst in der 7. Zeile, also in der Folgezeile feststellen, dass ein Syntaxfehler vorliegt. In der Zeile 6 fehlt nämlich eine abschließende Klammer.

Fehler bei Klammern passieren leider sehr häufig. Bei den meisten IDEs können Sie mithilfe des Editors prüfen, ob die Klammern richtig gesetzt sind. Klicken Sie auf eine öffnende Klammer, zeigt der Editor die schließende Klammer an.

Auch passiert es oftmals, dass Anführungszeichen und Hochkommata vergessen wurden.

```
Führe aus: 0805_335A4.py
  File "c:\users\gk\mu_code\0805_335a4.py", line 19
    print("Angebotspreis: , angebotspreis)
                                           ^
SyntaxError: EOL while scanning string literal
>>>
```

Die Fehlermeldung besagt, dass der Interpreter das Zeilenende (End Of Line) erkannt hat, während er einen String-Ausdruck untersucht.

Ein weiterer, häufiger Fehler ist der Schreibfehler bei den Variablen. Die stellt der Interpreter erst während des Programmlaufs fest.

```
12 vkpreis = zwischensumme + ust
13 print("Verkaufspreis: " + str(vkPreis) + " Euro")
14
```

```
Führe aus: 0805_4001 VK.py
Verkaufspreisberechnung
Eingabe Anzahl Liter: 2500
Eingabe Transportkilometer: 30
Traceback (most recent call last):
  File "c:\users\gk\mu_code\0805_4001 vk.py", line 13, in <module>
    print("Verkaufspreis: " + str(vkPreis) + " Euro")
NameError: name 'vkPreis' is not defined
```

Der Interpreter hat in dem Ausgabebefehl Zeile 13 festgestellt, dass die Variable *vkPreis* nicht existiert. Die Variable *vkpreis* wurde in Zeile 12 angelegt und mit einem Wert belegt. In der Ausgabeanweisung steht *vkPreis*. Und vk**P**reis und vkpreis sind für den Interpreter unterschiedliche Namen.

Ein anderer Fehlertyp ist der logische Fehler, der weiter unten besprochen wird.

Übung

Erweitern Sie das Programm in der Weise, dass der Nettopreis (hier: zwischensumme) und die Umsatzsteuer ausgewiesen werden.

3.2.2 Mathematische Funktionen

In dem obigen Programm *Verkaufspreisberechnung* werden für die Berechnung von Umsatzsteuer und Verkaufspreis nur 2 Nachkommastellen benötigt. Deshalb müssen die Werte in den Variablen oelPreis, transport und ust auf 2 Nachkommastellen gerundet werden.

Runden mit round()

Benötigt wird eine Rundungsfunktion. Python stellt mehrere Funktionen zur Verfügung, die man für das Runden brauchen könnte.

Eine Funktion ist ein Programm, das für eine spezifische Problemstellung geschrieben worden ist. Man unterscheidet Standardfunktionen von selbst programmierten Funktionen. Standardfunktionen werden auch built-in-Funktionen genannt, weil sie im Sprachumfang enthalten sind. Den anderen Funktionstyp muss der Programmierer selber schreiben.

Eine Funktion wird aus dem Anwenderprogramm aufgerufen. Ihr werden ein oder mehrere Werte übergeben, die sie verarbeiten soll. Abschließend gibt die Funktion den berechneten Wert an die aufrufende Stelle zurückgibt.

Beispiel

```
....
oelPreis = round(liter * literPreis, 2)                    # aufrufende Stelle
```

In dem Beispiel wird das Produkt aus *liter* und *literPreis* sowie die Anzahl der benötigten Nachkommastellen an die Funktion round() übergeben. Die Standardfunktion rundet das Produkt auf 2 Nachkommastellen und weist das Ergebnis der Variablen *oelPreis* zu.

Auch die in Python eingebauten Funktionen format() und math.floor könnten für das Runden verwendet werden, weil man mit ihnen Nachkommastellen manipulieren kann.

Runden mit floor()

Die mathematische Funktion floor(x) rundet auf die nächst kleinere Ganzzahl ab.

```
>>> math.floor(4585.987)
4585
>>>
```

Die Nachkommastellen werden abgeschnitten. Das Ergebnis muss aber 4585.99 sein. Deshalb muss die Zahl mit 100 multipliziert und anschließend durch 100 dividiert werden.

```
>>> math.floor(4585.987*100)/100
4585.98
```

Damit ist das korrekte Ergebnis noch immer nicht erreicht. Die dritte Dezimalstelle ist auch noch zu berücksichtigen. Deshalb wird sie um 0.5 erhöht.

```
>>> math.floor(4585.987*100+0.5)/100
4585.99
```

Die Rundungsanweisung lautet dann:

```
from math import *
...
gerundet = math.floor(zahl*100+0.5)/100
```

Die Funktion floor() befindet sich in der Mathematik-Bibliothek, die beim Start von Python nicht automatisch eingeladen wird. Deshalb muss man sie mittels des import-Befehls einbinden.

Runden mit format()

```
>>> format(4585.987,".2f")
'4585.99'
```

Die Formatangabe lautet ".2f". Damit liefert die Funktion format() einen Wert auf zwei Stellen gerundet. Dieser Wert ist aber keine Zahl, sondern ein String, zu erkennen an den Hochkommata. Somit muss der Wert mittels float() in eine reelle Zahl umformatiert werden.

```
gerundet= float(format(zahl, ".2f"))
```

Vorsicht: wenn die dritte Nachkommastelle eine 5 ist und keine weiteren Dezimalstellen folgen, wird ein falsches Ergebnis geliefert.

```
In[12]: format(45.985,'.2f')
Out[12]: '45.98'
```

Die Funktion format() dient zur formatierten Ausgabe von Zahlen in print()-Anweisungen. Es wandelt eine Zahl in eine Zeichenkette um. Für das Runden ist format() nicht geeignet. Das Runden unter Einsatz von floor() ist zu umständlich. round() ist die bessere Wahl.

Das Programm Verkaufspreisberechnung 0805_301 soll um die Rundungsfunktion ergänzt werden:

```
#Verkaufspreisberechnung mit Runden 0805_301b
#O805_301b VK

print ("Verkaufspreisberechnung")
literPreis = 0.527
kilometerPreis = 3.55
liter = int(input("Eingabe Anzahl Liter: "))
kilometer = int(input("Eingabe Transportkilometer: "))
oelPreis = round(liter * literPreis,2)
transport = round(kilometer *kilometerPreis,2)
zwischensumme = oelPreis + transport
ust = round(zwischensumme * 0.19, 2)
vkpreis = zwischensumme + ust
print("Verkaufspreis: " + format(vkpreis,'.2f') + " Euro")
```

Ergebnis

```
Verkaufspreisberechnung
Eingabe Anzahl Liter: 2500
Eingabe Transportkilometer: 30
Verkaufspreis: 1694.56 Euro

Process finished with exit code 0
```

3.2.3 Eigendefinierte Funktion

In Python ist es sehr einfach, eigene Funktionen zu definieren. Das sei an einem einfachen Beispiel erläutert. In dem Programm *Verkaufspreisberechnung* wird an mehreren Stellen eine Multiplikation von zwei Zahlen mit einer Rundung auf zwei Nachkommastellen verlangt. Das Runden soll nun in eine eigene Funktion ausgelagert werden. Die Funktion runden() könnte wie folgt definiert werden.

```
def runden(x,y):
    return round(x*y,2)


#Aufruf der Funktion
oelPreis = runden(liter, literPreis)
```

Die Funktion *runden()* erwartet zwei Werte, die sie multiplizieren und auf 2 Stellen runden soll. Nach Erledigung liefert sie den Wert an die Variable, die im Funktionsaufruf links vom Gleichheitszeichen steht.

Im Funktionsaufruf werden die Werte, die in den beiden Variablen stehen, an die Funktion übergeben. *Liter* → x, *literPreis* → y. Der Wert, den die Funktion zurückliefert, wird der Variablen *oelPreis* zugewiesen.

Vereinbarung einer Funktion

Die Funktionsvereinbarung wird eingeleitet mit

```
def funktionsname (formaler Parameter):
… anweisung                    } Funktions-
  anweisung                    } block
  return ergebnis
```

def ist ein Schlüsselwort. Der Funktionsname kann frei gewählt werden. Wird der Funktion ein Wert übergeben, so muss in runden Klammern eine Funktionsvariable als formaler Parameter angegeben sein. Die Vereinbarungszeile schließt ab mit einem Doppelpunkt. Nach dem

Anschlagen der Eingabetaste springt der Schreibcursor in der nächsten Zeile, versetzt um einen Tabulatorsprung. Diese Einrückung ist wichtig! Alle Anweisungen, die zu der Funktion gehören, müssen exakt untereinander eingerückt sein.

Die meisten Editoren sorgen mit einem **auto-indent,** dass die Folgezeilen eingerückt sind. Mit einem nochmaligen Anschlagen der Eingabetaste wird das auto-indent aufgehoben.

return

Die Funktion endet mit return ergebnis. Das Ergebnis wird an die aufrufende Stelle übermittelt.

Aufruf der Funktion

variable = funktionsname(aktueller Parameter)

Der aktuelle Parameter ist ein Argument, das der Funktion übergeben wird. Die Anzahl der formalen und aktuellen Parameter muss übereinstimmen.

An eine Funktion können kein, ein oder auch mehrere Argumente übergeben werden.

Beispiel 1

```
def strich():                                          # Vereinbarung ohne Parameter
    print("----------------------------------")


strich()                                               # Funktionsaufruf
```

In diesem Beispiel wird kein Parameter übergeben, es wird auch kein return benötigt, weil kein Wert zurückgegeben wird. Hier hat die Funktion strich() die Eigenschaft eines Unterprogramms.

Einen Unterstrich mit einer Länge von 30 erzeugt auch print('_'*30).

Beispiel 2

```
# Funktionsbeispiel2

def prozentrechnung(grundwert, prozentsatz):                    # Vereinbarung
    prozentwert = round(grundwert*prozentsatz/100,2)
    erhGrundwert = grundwert + prozentwert
    return prozentwert, erhGrundwert


ust, endpreis = prozentrechnung(100,19)                              # Aufruf

print('Umsatzsteuer: ', format(ust, '.2f'))
print('Endpreis:    ', format(endpreis,'.2f'))
```

Ergebnis

```
Umsatzsteuer:  19.00
Endpreis:     119.00
```

In diesem Beispiel werden zwei Werte an die Funktion übertragen, aber auch zwei Werte zurückgeliefert. In der Funktion wird sowohl der Prozentwert wie auch der erhöhte Grundwert berechnet und zurückgegeben. Der Inhalt von *prozentwert* wird an ust und der Wert in *erhGrundwert* an endpreis geliefert

Übung

Ergänzen Sie das Programm Verkaufspreisberechnung um die Rundungsfunktion, Rechenoperationen und Ausgabeanweisungen.

Die Ausgabe sollte wie folgt gestaltet sein:

```
Verkaufspreisberechnung
Eingabe Anzahl Liter: 2500
Eingabe Transportkilometer: 15.5
Ölpreis:        1317.50 Euro
Transportkosten:  55.02 Euro
-----------------------------------
Warenwert:      1372.52 Euro
Umsatzsteuer:    260.78 Euro
-----------------------------------
Verkaufspreis:  1633.30 Euro
```

Programmentwurf

```
#O805_302_VKrunden
def runden (zahl):
    return ???

def strich():
   print("-----------------------------------")

# Hauptprogramm
print ("Verkaufspreisberechnung")
literPreis = 0.527
kilometerPreis = 3.55
liter = int(input("Eingabe Anzahl Liter: "))
kilometer = float(input("Eingabe Transportkilometer: "))
....
```

3.2.4 Übersicht über mathematische Funktionen im Modul math

Funktionsname	Wirkung	Beispiel
floor(x)	Abschneiden von Nachkommastellen	floor(123.454) → 123
pow(x,y)	Exponentialrechnung x^y	pow(4,3) → 64
log(x)	Logarithmus von x zur Basis e	log(10) → 2.3025 …
sqrt(x)	Quadratwurzel von x für x > 0	sqrt(16) → 4

Beispiel

Alexander möchte wissen, wieviel Geld ihm nach 10 Jahren zur Verfügung steht, wenn er 3000,00 Euro auf ein Konto einzahlt, das mit 3 % verzinst wird.

Anfangskapital K_o = 3000, Zinssatz p% = 3 %, Anzahl der Jahre n = 10

Die Formel zur Berechnung des Endkapitals: $K_n = K_0 * q^n$ für q = 1 + p/100

```
#Endwertberechnung
#0805_324
def zinszins(kap, q,n):
    return kap*pow(q,n)

kapital = 3000
zinssatz = 3
jahre = 10
endkapital = zinszins(kapital, 1+zinssatz/100,jahre)
print('Kapital:  ' + format(endkapital,'.2f'))
```

3.2.5 Zufallszahlen

Python verfügt über ein gesondertes Modul, das sich mit Zufallszahlen befasst. Hier sind drei Funktionen von besonderem Interesse:

Funktionsname	Wirkung
choice(sequenz)	Zufallsauswahl aus einer Sequenz
random()	Zufällige Gleitkommazahl aus dem Intervall]0.0 , 1.0[
randint(a,b)	Zufällige Ganzzahl aus dem Intervall [a,b]

Beispiel

Der Zufallsgenerator sagt voraus, mit welchem Ergebnis das Spiel zwischen der Heimmannschaft und der Gastmannschaft endet. Die höchste Anzahl an Toren bzw. Sätzen beträgt 3.

```
# zufall
from random import *
heimTeam=randint(0,3)
gastTeam=randint(0,3)
print("Vorhersage: "+ str(heimTeam) + ":"+ str(gastTeam))
```

Ergebnis:

```
Vorhersage: 2:3
```

3.3 Zeichenketten und Zeichenkettenfunktionen – ein kurzer Einblick

Es gibt nicht nur Standardfunktionen für mathematische Problemstellungen, sondern auch für die Behandlung von Zeichenketten. Dieser Datentyp wird näher im Kapitel 6 behandelt. Weil aber in den beiden Folgekapiteln Zeichenketten angesprochen werden, erfolgt hier eine kurze Einführung.

Eine Zeichenkette (String) ist eine Abfolge von alphanumerischen Zeichen, die zwischen Anführungszeichen oder Hochkomma steht.

Beispiel

```
adresse =“45688 Wohnort Aastr. 32“
kundenname = ‘Herr Walter Waldmann‘
```

Stringverkettung – Konkatenation

```
postleitzahl =“54201“
ortsname = “Langstadt“
anschrift = postleitzahl + “ “ + ortsname
```

Mit dem Plus-Operator + werden Zeichenketten aneinander gehängt.

Der Parameter in der print-Anweisungen besteht aus verketteten Zeichenketten.

Beispiel

```
print('Kapital:  ' + format(endkapital,'.2f'))
```

Länge

Die Länge einer Zeichenkette liefert die Funktion len():

```
wortlaenge = len(“Langstadt“)        →      9
```

Zugriff auf einzelne Elemente

Die Darstellung einer Zeichenkette im Arbeitsspeicher kann sich wie folgt vorstellen:

ortsname	L	a	n	g	s	t	a	d	t

Index von vorne	0	1	2	3	4	5	6	7	8
Index von hinten	-9	-8	-7	-6	-5	-4	-3	-2	-1

Eine Zeichenkette wird in eine Folge von Bytes abgelegt. Der Index-Operator [] ermöglicht den Zugriff auf ein bestimmtes Element einer Zeichenfolge bzw. Sequenz. Die Zählweise beginnt mit Null. Der String ist also **nullbasiert**.

```
ortsname = “Langstadt“
print(ortsname[3])      → g
print(ortsname[3:6])    → gst
print(ortsname[-1])     → t
```

Die Anweisung ortsname[3:6] liefert einen zusammenhängenden Ausschnitt aus der Zeichenkette: gst

Eine Zeichenkette kann auch von hinten nach vorne bearbeitet werden. Zu beachten ist, dass die Zählweise von vorne nach hinten mit 0 beginnt. Die Zählweise von hinten nach vorne beginnt mit -1.

3.5 Datumsrechnung

Bei Datumsberechnungen muss das Modul date von der Bibliothek datetime importiert werden.

```
from datetime import *
sylvester =date(2022,12,31)
heute = date.today()
anzahlTage = sylvester – heute
print(anzahlTage)
```

Das Datum wird in der Form JJJJ,MM,DD angegeben.

3.6 Zusammenfassung

- Es gibt prinzipiell zwei Fehlerarten: Syntaxfehler und Logikfehler. Syntaxfehler stellt der Interpreter fest, Logikfehler muss der Programmierer selber herausfinden. Hilfreich ist das Debugging, die schrittweise Ausführung des Programms.
- Zum Runden von Dezimalzahlen wird die Standardfunktion round() angewendet: round(variable, Anzahl Nachkommastellen)
- Die Funktion format() wandelt eine Zahl in einen String mit einer bestimmten Anzahl von Nachkommastellen. Mittels format(variable,‘.2f‘) wird eine Dezimalzahl mit zwei Nachkommastellen ausgegeben.
- def funktionsname(formaler Parameter): definiert den Kopf einer eigendefinierten Funktion.
- Eine Zeichenkette (String) besteht aus einer Folge von alphanumerischen Zeichen, die in Anführungszeichen oder Hochkommata eingeschlossen ist.
- Zeichenketten werden mittels des Operators + aneinander gefügt.
- Der Zugriff auf eine bestimmte Stelle in der Zeichenkette erfolgt mittels des Operators [].
- Die Funktion print() macht nach jeder Ausgabe eine Zeilenschaltung. Um die Zeilenschaltung zu verhindert, wird als letztes Element end=““ gesetzt.

3.7 Aufgaben

Aufgabe 1

Der Rechnungsbetrag ergibt sich aus der Summe von Nettopreis und Umsatzsteuer. Entwickeln Sie ein Programm, das nach der Eingabe des Nettopreises und des Umsatzsteuersatzes die Umsatzsteuer und den Rechnungsbetrag ermittelt.

Aufgabe 2

Berechnung des Zinsertrages bei Termingeld:

Eingaben: Betrag, Zinssatz, Laufzeit

Ausgabe: Zinsertrag

Aufgabe 3

Auf welchen Betrag steigt eine Kapitaleinlage von 10000,00 Euro, die jährlich mit 2.5 % verzinst wird.

Formel: Endkapital = Anfangskapital * $(1 + \text{zinssatz}/100)^{\text{jahre}}$

Zusatz: Verwenden Sie unterschiedliche Zinssätze und vergleichen Sie die Endbeträge.

Aufgabe 4

Alexander will etwas für seine Altersversorgung tun. Er plant, 40 Jahre lang jeweils am Ende des Jahres 3000,00 Euro an eine Lebensversicherung zu zahlen. Die Versicherung gewährt einen Zinssatz von 3 %. Über welchen Betrag kann Alexander nach Ablauf von 40 Jahren verfügen?

Gesucht ist der Rentenendwert R_n.

Gegeben r = 3000, Zinssatz p = 3 %, Jahre n = 40

Die nachschüssige Rentenendwertformel lautet:

$$R_n = r*(q^n - 1) / (q - 1) \quad \text{für } q = 1 + p/100$$

Zusatz: Berechnen Sie die Rentenendwerte bei den Zinssätzen 2.5 %, 4.0 % und 6 % und vergleichen Sie die Endwerte**.**

Aufgabe 5

Der Angebotspreis ist zu berechnen. Der Handlungskostensatz beträgt 45 %. Der Gewinn ist mit 10 % angesetzt. Dem Kunden werden 2 % Skonto eingeräumt.

Schreiben Sie das Programm, mit dem der Angebotspreis berechnet wird.

Eingabe: Einstandspreis

Ausgabe:

```
Einstandspreis: 100
Selbstkostenpreis:  145.00
-------------------------
Barverkaufspreis:   159.50
-------------------------
Angebotspreis:      162.76
```

Aufgabe 6

Die Mitarbeiter reichen nach einer mehrtägigen Geschäftsreise ihre Belege für Übernachtung und Frühstück ein.

3 x Übernachtung: 330,00

3 x Frühstück : 90,00

Gesamt: 420,00

Darin sind 58,58 Euro Umsatzsteuer enthalten.

Mirjam arbeitet in der Buchhaltung. Sie hat die Aufgabe, den Umsatzsteuerbetrag aufzuschlüsseln nach 7 % und 19 %. Die Übernachtungskosten enthalten 19 % Umsatzsteuer.

4 Auswahlstruktur

4.1 Auswahlstruktur mit einfacher Bedingung

Problem

In der Firma Klender hängt der Ölpreis von der Menge ab, die ein Kunde kauft. Ab einer Liefermenge von 2000 Litern beträgt der Literpreis 0,527 € anstelle 0,557 €.

Bei einer solchen Problemstellung ist das im Abschnitt 3 behandelte Programm nur noch bedingt einsetzbar. Es muss um eine Auswahlstruktur erweitert werden.

Programme mit Auswahlstruktur enthalten Anweisungen, die nur dann ausgeführt werden, wenn eine bestimmte Bedingung erfüllt ist.

Eine Bedingung besteht aus einem Vergleich.

Beispiel

a = 3, b = 4

Bedingung	Bedingungsausdruck	Ergebnis
gleich	a == b	False
größer	a > b	False
kleiner	a < b	True
größer oder gleich	a >= b	False
kleiner oder gleich	a <= b	True
ungleich	a != b	True

Hinweis: Bei einem Vergleich auf Gleichheit muss ein **doppeltes** Gleichheitszeichen geschrieben werden, da das einfache Gleichheitszeichen als Zuweisungsoperator dient.

Zur Steuerung der Auswahl stehen verschiedene Formen von Auswahlanweisungen zur Verfügung:

- einseitige Auswahl
- zweiseitige Auswahl
- mehrseitige Auswahl

4.1.1 Zweiseitige Auswahl

Beispiel

Bei der Berechnung des Verkaufspreises ist zu berücksichtigen, dass bei einer Abnahmemenge von weniger als 2000 Liter der Literpreis 0,557 Euro statt 0.527 Euro beträgt.

Welcher Preis anzuwenden ist, hängt ab von der Literanzahl.

Wenn liter < 2000, dann wende 0.557 € als literPreis an. Ansonsten verwende 0.527 € als literPreis, um den Ölpreis zu berechnen.

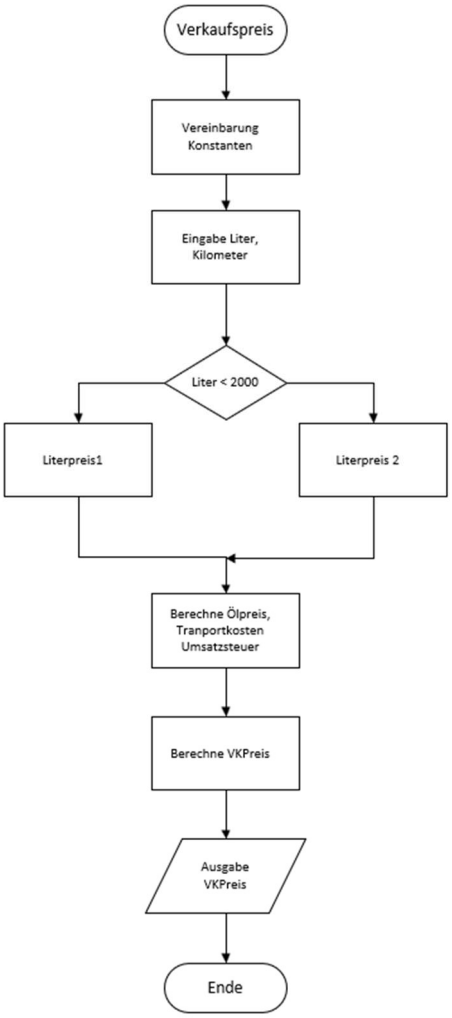
Es liegt hier eine Wenn-Dann-Sonst-Struktur vor. Ist die Wenn-Bedingung erfüllt ist, wird die Anweisung auf dem Dann-Weg bearbeitet, ansonsten die Anweisung auf dem Sonst-Weg.

Um sich die Logik zu verdeutlichen, kann man den Pseudo-Code oder auch ein Struktogramm einsetzen:

Pseudo-Code

```
Wenn Literanzahl < 2000
        literPreis = 0,557
Sonst
        literPreis = 0.527
berechne oelPreis
```

In Abhängigkeit von der Literzahl wird *literPreis1* oder *literPreis2* angewendet.



Programmablaufplan

Der Programmablaufplan zeigt die Programmlogik. Nach der Vereinbarung der Konstanten für *literPreis1*, literPreis2 und *kilometerPreis* erfolgt die Eingabe für *liter* und *kilometer*. In Abhängigkeit vom Wert in *liter* wird der Variablen *literPreis* entweder *literPreis1* oder *literPreis2* zugewiesen. Anschließend erfolgen die Berechnungen und die Ausgabe des Verkaufspreises

Programm

```
#Verkaufspreisberechnung
0805_411_1
#0805_411_1
def runden(zahl):
    return round(zahl,2)

print ("Verkaufspreisberechnung")
literPreis1= 0.557                                  # Preis bei weniger als 2000 l
literPreis2 = 0.527
kilometerPreis = 3.55
liter = int(input("Eingabe Anzahl Liter: "))
kilometer = int(input("Eingabe Transportkilometer: "))
if liter < 2000:
```

```
   literPreis = literPreis1                                    # Dann-Block
else:
   literPreis = literPreis2                                    # Sonst-Block
oelPreis = runden(liter * literPreis)
transportKosten = runden(kilometer *kilometerPreis)
zwischensumme = oelPreis + transportKosten
ust = runden(zwischensumme * 0.19)
vkpreis = zwischensumme + ust
print("Verkaufspreis: " + format(vkpreis, '.2f') + " Euro")
```

Erläuterung

Der Bedingungsausdruck wird mit **if** eingeleitet. Dahinter wird die Bedingung formuliert. Die Bedingungszeile schließt mit einem Doppelpunkt.

Das Ergebnis eines Bedingungsausdrucks ist entweder wahr oder falsch.

Ist der Ausdruck wahr, wird der Dann-Anweisungsblock ausgeführt, der in der Folgezeile notiert wird. Die Zeile bzw. Zeilen müssen eingerückt sein. Meist steuert der Editor die Texteingabe schon korrekt. Auf jeden Fall ist die Einrückung wichtig!

Das Schlüsselwort **else:**, das in derselben Fluchtlinie wie das **if** stehen muss, leitet den Sonst-Anweisungsblock ein. Dieser wird bearbeitet, wenn der Bedingungsausdruck falsch ist. Dieser Anweisungsblock muss auch eingerückt geschrieben werden.

Wird die Einrückung aufgehoben, ist die if-Anweisung beendet. Ein erneuter Anschlag der Eingabetaste setzt den Schreibcursor an den Zeilenanfang.

Aufbau einer zweiseitigen Auswahl

```
if Bedingungsausdruck:
    Anweisung1
    Anweisung2
       ...
    Anweisung n
else:
    Anweisung1
    Anweisung2
    ...
    Anweisung n
Folgeanweisung
```

Python arbeitet mit einer Blockstruktur. Alle Anweisungen, die zu einem Block gehören, müssen in derselben Flucht stehen, also entsprechend eingerückt sein.

Eingeschachtelte Auswahl

Problem

Der Öllieferant kalkuliert den Verkaufspreis nach folgendem Schema:

Liter < 2000
: Bei einer Entfernung ab 5 Kilometer erhöhen sich die Transportkosten um 10 %.

Liter >= 2000
: Bei einer Entfernung ab 10 Kilometer erhöhen sich die Transportkosten um 5 %.

Pseudo-Code

```
Wenn liter < 2000
    literPreis = literPreis1
    Wenn kilometer < 5
            transportkosten = kilometer * kilometerPreis
    Sonst
            transportkosten = kilometer * kilometerPreis * 1.1
Sonst
    literPreis = literPreis2
    Wenn kilometer < 10
            transportkosten = kilometer * kilometerPreis
    Sonst
            transportkosten = kilometer * kilometerPreis * 1.05
berechne oelPreis
```

```
#Verkaufspreisberechnung 0805_411_2
#0805_411_2
def runden(zahl):
   return round(zahl,2)

print ("Verkaufspreisberechnung")
literPreis1= 0.557                                  # Preis bei weniger als 2000 l
literPreis2 = 0.527
kilometerPreis = 3.52
liter = int(input("Eingabe Anzahl Liter: "))
kilometer = int(input("Eingabe Transportkilometer: "))
```

```
if liter < 2000:
    literPreis = literPreis1
    if kilometer <5:
        transportKosten=runden(kilometer *kilometerPreis)
    else:
        transportKosten=runden(kilometer *kilometerPreis*1.1)

else:
    literPreis = literPreis2
    if kilometer <10:
        transportKosten = runden(kilometer *kilometerPreis)
    else:
        transportKosten = runden(kilometer *kilometerPreis*1.05)

oelPreis = runden(liter * literPreis)
zwischensumme = oelPreis + transportKosten
ust = runden(zwischensumme * 0.19)
vkpreis = zwischensumme + ust
print("Verkaufspreis: " + format(vkpreis,'.2f') + " Euro")
```

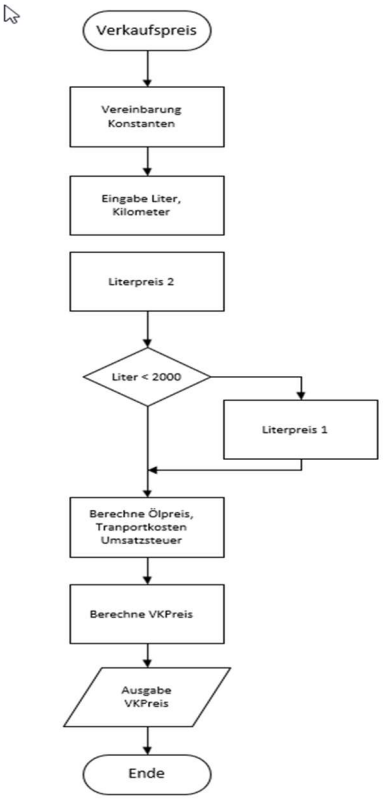
Bei einer einschachtelten if-Anweisung ist ebenfalls auf die Einrückungen zu achten. Nur aufgrund der Einrückungen kann der Interpreter erkennen, was zusammengehört.

4.1.2 Einseitige Auswahl

Oftmals kann man durch eine Vorbesetzung eine zweiseitige Auswahl vermeiden und das Problem mit einer einseitigen Auswahl lösen.

Beispiel

```
...
liter = int(input("Eingabe Anzahl Liter: "))
kilometer = int(input("Eingabe Transportkilometer: "))

literPreis = literPreis2                                    # Vorbesetzung
if liter < 2000:
    literPreis = literPreis1

oelPreis = runden(liter * literPreis)
```



Es wird hier angenommen, dass in der Regel mehr als 2000 Liter geliefert werden. Deshalb wird *literPreis* mit dem Wert von *literPreis2* vorbesetzt. Dann braucht nur noch geprüft zu werden, ob die Liefermenge geringer ist als 2000.

Aufbau einer einseitigen Auswahl

```
if Bedingungsausdruck:
    Anweisung1
    Anweisung2
       ...
    Anweisung n

Folgeanweisung
```

Bei der einseitigen Auswahl entfällt der else-Weg. Der ist nicht erforderlich, da vor der if-Anweisung die Bedingungs-variable mit einem der beiden möglichen Werte vorbesetzt wurde.

4.1.3 Mehrseitige Auswahl

Problem

Zur Berechnung des Ölpreises ist eine Preisstaffel zu berücksichtigen. Der Literpreis hängt ab von der Bestellmenge.

< 2000 Liter → 0.557 Euro
< 5000 Liter → 0.527 Euro
< 8000 Liter → 0.5 Euro
>= 8000 Liter → 0.495 Euro

Für die mehrseitige Auswahl stellt Phyton eine *if-elif-else*-Anweisung zur Verfügung. ***elif*** ist eine Abkürzung für else-if.

Programm

```
literPreis1= 0.557          #Preis bei weniger als 2000 l
literPreis2 = 0.527         # Preis für >=2000 und < 5000
literPreis3 = 0.5           # Preis für >=5000 und < 8000
literPreis4 = 0.495         # Preis für >= 8000
kilometerPreis = 3.52
liter = int(input("Eingabe Anzahl Liter: "))
kilometer = int(input("Eingabe Transportkilometer: "))
if liter < 2000:
   literPreis = literPreis1
elif liter < 5000:
   literPreis = literPreis2
elif liter < 8000:
   literPreis = literPreis3
else:
   literPreis = literPreis4
oelPreis = runden(liter * literPreis)
```

Hinter elif wird die Bedingung in der gleichen Weise formuliert wie hinter if.

Der *else*-Block ist optional. Das bedeutet, dass er formuliert werden kann, aber nicht zwingend erforderlich ist. Wenn er benötigt wird, muss er am Ende der if-Anweisung stehen.

Aufbau einer Mehrfachauswahl

Die if-Anweisung mit einer elif-Klausel wird allgemein wie folgt gestaltet:

```
if bedingung_1:
        Anweisungsblock
elif bedingung_2:
        Anweisungsblock
elif bedingung_3:
        Anweisungsblock
...
[else:
        Anweisungsblock_n]          #optional
```

4.1.4 Vergleich bei Zeichenketten

Beim Vergleich von Zeichenketten bzw. Strings gelten die oben behandelten Vergleichsoperatoren. Die Gleichheitsrelation wird ebenfalls durch ein doppeltes Gleichheitszeichen dargestellt.

Problem: Ist Alexander größer als Alexandra?

```
#0805_414 String-Vergleich
#0805_414
person1 = "Alexander"
person2 = "Alexandra"
if  person1 < person2:
   print("das liegt am Alphabet")
else:
   print('der ASCII-Code von B ist 66, von A ist er 65')
print('Ausgabe im ASCII-Format')
for zeichen in person1:
   print(zeichen, zeichen.encode('utf-8').hex())
print()
for zeichen in person2:
   print(zeichen, zeichen.encode('utf-8').hex())
```

Beim Vergleich von Zeichenketten wird der ASCII-Code Stelle für Stelle miteinander verglichen. „B“ ist größer als „A“, weil der ASCII-Code für den Buchstaben „B“ größer ist als der von „A“.

ASCII-Code		Erläuterung	
A 41	A 41	0010 0001	Byte-Struktur
l 6c	l 6c	0110 1100	
e 65	e 65		
x 78	x 78		
a 61	a 61		
n 6e	n 6e	Bei dem Buchstaben „e“ entscheidet es sich, dass „Alexander“ kleiner ist als „Alexandra“.	
d 64	d 64		
e 65	r 72		
r 72	a 61		

4.2 Auswahlstruktur mit Bedingungsgefüge

4.2.1 Bedingungsgefüge

Ein Bedingungsgefüge besteht aus mehreren Bedingungen, die miteinander verknüpft sind. Die Verknüpfung kann durch UND, ODER oder Exclusive ODER erfolgen. Die entsprechenden englischen Begriffe lauten AND, OR und XOR.

Verknüpfungsregeln der UND-Verknüpfung (Konjunktion)

Bedingung 1	Bedingung 2	Ergebnis der Konjunktion
A = True	B = True	A ^B = True
A = False	B = True	A ^B = False
A = True	B = False	A ^B = False
A = False	B = False	A ^B = False

Verknüpfungsregeln der ODER-Verknüpfung (Disjunktion)

Bedingung 1	Bedingung 2	Ergebnis der Disjunktion
A = True	B = True	A v B = True
A = False	B = True	A v B = True
A = True	B = False	A v B = True
A = False	B = False	A v B = False

Verknüpfungsregeln der Exclusiv- ODER-Verknüpfung

Bedingung 1	Bedingung 2	Ergebnis der Kontravalenz
A = True	B = True	A v̲ B = False
A = False	B = True	A v̲ B = True
A = True	B = False	A v̲ B = True
A = False	B = False	A v̲ B = False

Das Ergebnis der XOR-Verknüpfung ist dann wahr, wenn nur eine der beiden Bedingungen wahr ist.

NOT

Eine weitere Verknüpfungsart ist die Verneinung NOT:

not (3 >4) → True

4.2.2 AND, OR, NOT, bedingte Ausdrücke

Problem

Die Verkaufspreisberechnung erfährt eine Änderung. Die Mindestmenge beträgt 1000 Liter, die Höchstmenge 12000 Liter. Außerhalb dieser Mengen wird kein Auftrag angenommen.

Lösung

```
if liter <= 1000 or liter >12000:
   print("keine Auftragsannahme")
else:
   print(“Auftrag wird angenommen“)
```

Diese Problemstellung lässt sich auch mit folgendem logischen Ausdruck, wie er in der Mathematik üblich ist, lösen:

```
if 12000 >= liter >= 1000:
   print("Auftrag angenommen")
else:
   print("keine Auftragsanahme"
```

Diese Schreibweise übersetzt Python in eine AND-Verknüpfung:

```
if liter >= 1000 and liter <= 12000:
    print(“Auftrag angenommen“)
else:
   print (“keine Auftragsannahme“)
```

Problem

Beträgt die Bestellmenge mehr als 10000 Liter und liegt die Transportentfernung unterhalb von 5 Kilometern, werden keine Transportkosten berechnet.

Lösung

```
if kilometer < 5 and liter >10000:
   transportKosten = 0
else:
   transportKosten = round(kilometer *kilometerPreis,2)
zwischensumme = oelPreis + transportKosten
```

Bedingte Ausdrücke

Python gestattet eine rationelle Formulierung von Bedingungsausdrücken. In dem obigen Beispiel wurde der Wert von *literPreis* durch die Konstruktion

```
literPreis = literPreis2                                    # Vorbesetzung
if liter < 2000:
   literPreis = literPreis1
```

bestimmt. In Python kann man die Auswahl des Literpreises auch wie folgt formuliert:

```
literPreis = (literPreis1 if liter < 2000 else literPreis2)
```

Aber auch komplexe Bedingungsausdrücke lassen sich in einer Programmzeile formulieren:

```
transportKosten=(0 if kilometer<5 and liter > 10000 else
                                                    round(kilometer*kilometerPreis,2))
```

Das Konstruktionsprinzip ist: Wert A if Bedingung erfülllt sonst Wert B.

Auch bei der print-Anweisung sind bedingte Ausdrücke zulässig.

Beispiel

```
titel = 'Herr Professor'
print("Herr" if titel == ' ' else titel)
```

Ausgabe: Herr Professor

4.3 Zusammenfassung

- Eine Auswahlstruktur wird durch if-else bzw. if-elif – else gesteuert.
- In eine Auswahlstruktur können weitere eingeschachtelt werden.
- Welcher Anweisungsblock ausgeführt wird, hängt ab vom Ergebnis einer Bedingungsprüfung.
- Das Ergebnis einer Bedingungsprüfung ist stets wahr oder falsch.
- Eine Bedingung kann aus einem einfachen Bedingungsausdruck oder auch aus einem komplexen Bedingungsgefüge bestehen.
- Der einfache Bedingungsausdruck besteht aus einem Kleiner-/Größer-/Gleich-Vergleich von zwei Werten.
- Bei komplexen Bedingungsgefügen werden mehrere Vergleichsergebnisse durch AND, OR oder NOT verbunden.

- Man unterscheidet einseitige, zweiseitige sowie mehrseitige Auswahlstrukturen.
- Beim Vergleich von Zeichenketten werden die ASCII-Werte der Zeichen verglichen.
- Bedingte Ausdrücke verkürzen den Programmtext zu Lasten der Lesbarkeit.

4.4 Aufgaben

Aufgabe 1

In der Firma Klender beträgt laut Betriebsvereinbarung die durchschnittliche monatliche Arbeitszeit 165 Stunden. Leistet ein Arbeiter mehr, werden die Überstunden mit 10 % vergütet. Zu Berechnung des Bruttolohns wurde das nachstehende Programm entwickelt. Es ist leider fehlerhaft. Was ist falsch?

```
# 0805_44A1 Fehlersuche
stunden = int(input("Eingabe Arbeitsstunden: "))
lohnsatz = int(input("Eingabe Stundenlohn: "))
if stunden > 165
   gesStdLohn = stunden* lohnsatz
   uebStdLohn = (stunden - 165)*lohnsatz*1.1
   bruttoLohn = round(gesStdLohn + uebStdLohn,)
print("Bruttolohn 1:      "+  format(gesStdLohn,'.2f'))
print ("Ueberstundenlohn: " + format(uebStdLohn,'.2f'))
print ("Bruttolohn 2:     " + format(bruttoLohn,'.2f'))
else:
bruttoLohn = round(stunden * lohnsatz,2)
   print("Bruttolohn:       " + format(bruttoLohn,'.2f'))
```

Aufgabe 2

Überarbeiten Sie das Programm von Aufgabe 1 in der Weise, dass eine einfache if-Anweisung genügt. Das Runden sollte in eine Funktion ausgelagert werden.

Aufgabe 3

Tagesgeldanlage: Eine Bank bietet folgende Konditionen an.

Anlagedauer in Tagen	Zinssatz
<=180	1.25
> 180	1.5

Erstellen Sie ein Programm, mit dem die Zinsen berechnet werden.

Eingabe: Anlagedauer, Zinssatz

Ausgabe: Zinsen (gerundet auf 2 Nachkommastellen)

Aufgabe 4

Ein Kreditinstitut hat folgende Konditionen für Festgeldanlage:

Anlagedauer in Tagen	Anlagebetrag	Zinssatz
= 60	>=120000	1.75
= 60	<120000	1.5
= 90	>= 100000	3
= 90	< 100000	2

Eingabe und Ausgabe wie bei Aufgabe 3

Aufgabe 5

Punkte	Notenbezeichnung
0	ungenügend
1, 2, 3	mangelhaft
4,5,6	ausreichend
7,8,9	befriedigend
10,11,12	gut
12,14,15	sehr gut

Die Punktestufen in der gymnasialen Oberstufe reichen von 0 bis 15. Ihnen sind die Notenbezeichnungen gem. obiger Tabelle zugeordnet.

Nach der Eingabe der Punkteziffer soll die Notenbezeichnung ausgegeben werden.

Aufgabe 6

Für die Betriebsabrechnung müssen die Energiekosten erfasst werden. Die Verbräuche werden von Wasser-, Gas- und Stromzählern erfasst. Der Wasserzähler kann maximal 999 cbm anzeigen. Dann beginnt er wieder von 0 an zu zählen. Die Höchstanzeige des Gaszählers beträgt 9999 cbm. Der Stromzähler hat den Übersprung bei 99999 kwh.

Eingaben: Art des Zählers: w = Wasser-, g = Gas-, s = Stromzähler
Alter Zählerstand,
neuer Zählerstand

Ausgabe:

```
Verbrauchsberechnungen
Eingabe Geräteart(w= Wasser. g = Gas. s = Stromzaehler):
Eingabe alter Zählerstand: 1266
Eingabe neuer Zählerstand: 253
Strom-Verbrauch: 98987.00
```

Entwickeln Sie einen geeigneten Algorithmus und schreiben Sie das Programm.

Aufgabe 7

Geschwindigkeitsüberschreitung – das kann teuer werden. Schreiben Sie ein Programm, dass die Geldstrafe, die Punkte und die Dauer des Fahrverbotes ausgibt nach der Eingabe von

- Geschwindigkeit,
- innerorts bzw. außerorts

Geschwindigkeitsüberschreitung innerorts (Pkw) – Buß

Geschwindigkeitsüberschreitung innerorts	Bußgeld	Punkte
bis 10 km/h	30 €	
11 - 15 km/h	50 €	
16 - 20 km/h	70 €	
21 - 25 km/h	115 €	1
26 - 30 km/h	180 €	1

Geschwindigkeitsüberschreitung außerorts	Bußgeld	Punkte	Fahrverbot
bis 10 km/h	20 €		kein Fahrverbot
11 - 15 km/h	40 €		kein Fahrverbot
16 - 20 km/h	60 €		kein Fahrverbot
21 - 25 km/h	100 €	1	kein Fahrverbot
26 - 30 km/h	150 €	1	1 Monat*

Aufgabe 8

Das Random-Beispiel (Abschnitt 3.2.5) soll nach der Bildung der Zufallszahlen anzeigen, welche Mannschaft gewonnen hat bzw. ob unentschieden gespielt wurde.

Aufgabe 9

Fortführung von Aufgabe 8:

Das Random-Programm soll zu einem Wett-Programm erweitert werden. Der Zufallszahlengenerator legt fest, wie viele Tore die Heim- und die Gastmannschaft erzielen.

Sie geben ebenfalls die Anzahl der Tore für die beiden Mannschaften ein.

Stimmen die Werte überein, erhalten Sie 3 Punkte. Einen Punkt erhalten Sie, wenn Sie den Sieg bzw. die Niederlage richtig getippt haben.

Beispiel: Ihre Eingabe: Heimmannschaft: 3, Gastmannschaft: 2

Computer gibt den Endstand mit 2:1 an

In den anderen Fällen gibt es keinen Punkt.

Ausgabebeispiel:

```
wie viele Tore schießt Bayern München?: 3
wie viele Tore schießt Manchester United?: 2
Sieg München vorhergesagt
Ergebnis: 2:0
Dein Tipp: 3:2
Punkte:1
```

5 Wiederholungsstrukturen

Problem 1

Alexandra möchte sich einen wunderschönen Scooter kaufen. Er soll 3000,00 Euro kosten. Von ihrem schmalen Azubi-Gehalt kann sie jeden Monat 100,00 Euro zurückzahlen. Der Verkäufer erklärt sich einverstanden und ergänzt: „Den Restschuld musst du mit 5 % verzinsen." Alexandra: „Wie soll ich das verstehen? Verkäufer: „Ich rechne dir das mal vor."

Kredit : 3000		Tage=30	Zinssatz	= 5%
Datum	Restschuld	Rate	Zinsen	Restschuld
01.08.	3000,00 €		12,33 €	3.012,33 €
01.09.	3.012,33 €	100,00 €	11,97 €	2.924,30 €
01.10.	2.924,30 €	100,00 €	11,61 €	2.835,90 €

Alexandra: „Und wann habe ich alles bezahlt?" Verkäufer: „Du brauchst die Rechnung nur solange fortsetzen, bis die Restschuld 0,00 ist. Das Rechenverfahren bleibt bei jeder Zahlung gleich."

Problem 2

Alexandra ärgert sich. Sie hat versehentlich bei der Eingabe der Ölmenge 150 statt 1500 eingegeben. Diese geringe Menge ist gar nicht zulässig, da die Mindestmenge 1000 Liter beträgt.

Bei vielen Aufgabenstellungen besteht die Notwendigkeit, dass eine Gruppe von Anweisungen wiederholt auszuführen ist. Es wäre aber viel zu umständlich, diesen Anweisungsblock so oft in das Programm zu schreiben, wie er zu wiederholen ist. Oftmals weiß der Programmierer auch nicht im Vorhinein, wie oft der Anweisungsblock auszuführen ist.

Es wird also ein Anweisungstyp benötigt, der in Abhängigkeit von einer Bedingung die wiederholte Ausführung von Anweisungen steuert.

In Python kann die Wiederholung bzw. die Iteration durch eine while- oder eine for-Anweisung gesteuert werden

5.1 WHILE-Anweisung

Mittels der while-Anweisung wird ein Block von Anweisungen wiederholt ausgeführt, solange eine Bedingung erfüllt ist.

Grundschema einer while-Anweisung:

```
....
anweisung
while Bedingung:          Schleifeneintrittsbedingung
      anweisung
      anweisung           } Schleifenkörper
      anweisung
      ....
anweisung
....
```

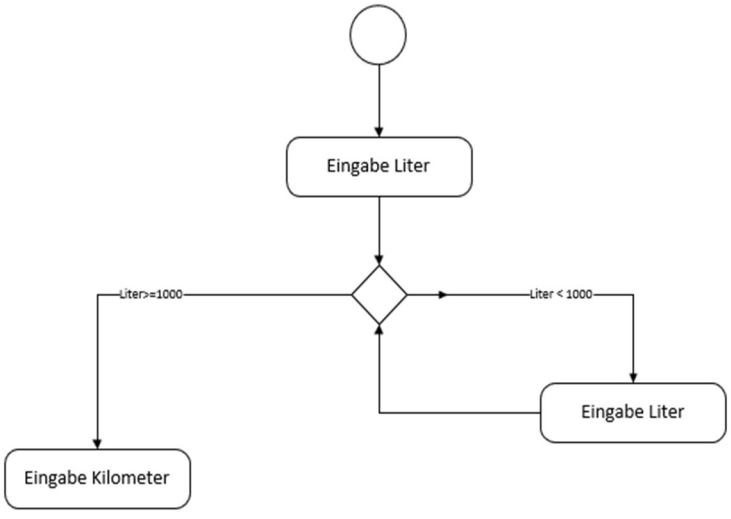
5.1.1 Steuerung von Eingaben

In dem Programm Verkaufspreisberechnung ist zu berücksichtigen, dass die Mindestabnahmemenge 1000 Liter beträgt. Wird ein geringere Menge eingegeben, liegt ein Eingabefehler vor. Ein solcher Fehler muss „abgefangen" werden.

Aufgrund dieser Anforderung muss die Eingabe des Betrages in eine Schleifenkonstruktion eingebaut werden. Umgangssprachlich: Solange die Literangabe kleiner als 1000 ist, muss die Eingabe wiederholt werden.

UML- Diagramm

Das UML-Diagramm verdeutlicht die Ablauflogik.



Programm

Das Python-Programm gestaltet sich wie folgt:

```
....
liter = int(input("Eingabe Anzahl Liter: "))                    # Vor-Lesen
while liter < 1000:
    print("Unzulässige Literangabe, mindestens 1000 Liter!")
    liter = int(input("Eingabekorrektur Liter:  "))              # Nach-Lesen
kilometer = int(input("Eingabe Transportkilometer: "))
....
```

Hinter while wird die Schleifeneintrittsbedingung formuliert: liter < 1000.

Nach der Eingabe der Zahl prüft das Programm, ob die Literanzahl kleiner als 1000 ist. Ist das nicht der Fall, überspringt das Programm den Schleifenblock und fährt mit der Eingabe der Kilometer fort.

Wurde ein Zahl kleiner als 1000 eingegeben, ist die Schleifeneintrittsbedingung erfüllt. In der nächsten Anweisung wird auf den Fehler hingewiesen und zur erneuten Eingabe aufgefordert. Nach der Eingabe prüft das System, ob die Schleifeneintrittsbedingung erfüllt ist. Wurde wiederum eine Zahl kleiner 1000 eingetippt, wird die Schleife erneut durchlaufen.

Die Schleife wird erst dann verlassen, wenn die eingegebene Zahl 1000 oder größer als 1000 ist.
Wichtig: Die Variable in dem while-Bedingungsausdruck muss mit einem Wert vorbelegt sein. Deshalb steht in dem obigen Beispiel die Eingabeanweisung vor der while-Anweisung. Für die weiteren Eingaben ist der Eingabebefehl am Ende der Schleife zuständig.

Diese Technik bezeichnet man auch als Vor-Lesen und Nach-Lesen. Dadurch wird sichergestellt, dass die Bedingungsvariable einen sinnvollen Wert enthält.

5.1.2 Tabellen erstellen

Problem

Ein neuer Tanklastwagen soll beschafft werden, Preis: 125.000,00 Euro. Die Frage ist, in wie vielen Jahren ist er auf einen Restwert von 10.000 Euro abgeschrieben, wenn man einen Abschreibungssatz von 30 % ansetzt?

Erforderliche Eingaben:

- Anschaffungswert,
- Abschreibungsprozentsatz,
- Restwert

Ausgabe:

```
Anschaffungswert: 125000
Abschreibungssatz: 30
Restwert: 10000
Jahr  Abschreibung   Buchwert
------------------------------
 1      37500.00      87500.00
 2      26250.00      61250.00
 3      18375.00      42875.00
 4      12862.50      30012.50
 5       9003.75      21008.75
 6       6302.62      14706.13
 7       4411.84      10294.29
 8        294.29      10000.00
```

Programm

```
# Wiederholungsanweisung - Degressive Abschreibung
#0805_512
def runden(zahl):
    return round(zahl,2)

def ausgabe(j, a, b):
    print(format(j,'2.0f')+format(a, '14.2f')+ format(b, '14.2f'))

anschaffungswert = float(input("Anschaffungswert: "))
abschreibungssatz = float(input("Abschreibungssatz: "))
restwert = float(input("Restwert: "))
jahr = 1
abschreibung = runden(anschaffungswert*abschreibungssatz/100)
buchwert = runden(anschaffungswert - abschreibung)                    #1
print ("Jahr  Abschreibung   Buchwert")
print ("-----------------------------------")
while buchwert  >= restwert:                                          #2
    ausgabe(jahr, abschreibung, buchwert)                             #3
    abschreibung = runden(buchwert*abschreibungssatz/100)
    buchwert = buchwert - abschreibung
    jahr += 1
ausgabe(jahr, buchwert + abschreibung -restwert, restwert)            #4
```

Erläuterungen

Wie im vorhergehenden Abschnitt dargestellt, müssen die Variablen, die in dem while-Bedingungsausdruck aufgeführt sind, vorher mit einem der Aufgabenstellung entsprechenden Wert belegt werden. In dem Bedingungsausdruck wird der Buchwert mit dem Restwert abgeglichen. Somit muss die Variable *Buchwert* vor Eintritt in die Schleife einen Wert aufweisen (#1).

In dem obigen Programm wurden für das 1. Jahr die Abschreibung und der Buchwert berechnet. Das erlaubt dann eine sinnvolle Auswertung des Bedingungsausdrucks (#2). Somit können auch in der ersten Anweisung in der while-Schleife die Werte für das erste Jahr der Abschreibung ausgegeben werden (#3)

Die Berechnungen für die Folgejahre werden in den drei weiteren Anweisungen vorgenommen. Ist die Schleifeneintrittsbedingung erfüllt, werden die Jahreswerte ausgedruckt.

Ist der Buchwert geringer als der Restwert, wird die Schleife verlassen. Der Buchwert weist bei einer prozentualen Abschreibung noch einen Wert auf. Deshalb wird der restliche Abschreibungsbetrag für das letzte Abschreibungsjahr nach der Schleife ausgegeben (#4).

Spaltengenaue Ausgabe

Die Ausgabe erfolgt über die Funktion *ausgabe(j, a, b)*. Bei einer Ausgabe in eine Tabelle müssen die Werte stellengerecht untereinander stehen. Das erreicht man mittels der Formatanweisung

format(var, ‘Vorkommastellen.Nachkommastellen‘)

Die Ausgabe der Werte wird durch ‘14.2f‘ formatiert. Somit wird die Zahl mit 14 Vorkomma- und zwei Nachkommastellen angezeigt. Dadurch erreicht man eine spaltengenaue Ausrichtung der Ausgabe.

5.1.3 While True – continue - break

Bei komplexen Schleifeneintrittsbedingungen besteht die Gefahr, dass die Programmlogik nur noch schwer zu durchschauen ist. Abhilfe bringt hier eine *while-True – if* – Kombination.

Bei while True als Schleifeneintrittsbedingung erfolgt ein Eintritt in die *while*-Schleife ohne weitere Bedingung. Innerhalb des Schleifenkörpers wird mittels *if*-Anweisung geprüft, ob noch ein weiterer Schleifendurchlauf erforderlich ist. Falls nicht, wird durch die *break*-Anweisung die Schleife verlassen.

Andernfalls wird mittels der Anweisung *continue* die Schleifenbearbeitung fortgesetzt.

Die Schleifensteuerung wird also in das Innere der Schleife verlagert.

Beispiel

In dem Abschreibungsprogramm muss sichergestellt sein, dass der Anschaffungswert eine positive Zahl ist. Die Eingabe einer negativen Zahl soll abgefangen werden. Dazu dient die folgenden Eingaberoutine.

```
while True:
    anschaffungswert = float(input("Anschaffungswert: "))
    if anschaffungswert < 0:
        print ("Der Anschaffungswert kann nicht negativ sein!")
        continue
    else:
        break
abschreibungssatz = float(input("Abschreibungssatz: "))
```

Mit *while True* ist die Schleifeneintrittsbedingung in jedem Fall erfüllt. Innerhalb der Schleife erfolgt die Eingabe, die anschließend mittels der *if*-Anweisung überprüft wird. Ist der eingegebene Wert negativ, erfolgt der Hinweis. Anschließend kann der Anschaffungswert erneut eingetippt werden. Dafür sorgt die Anweisung ***continue***, die zurück zum Schleifenanfang führt.

Wurde eine positive Zahl eingegeben, wird der *else*-Weg beschritten. Die Anweisung ***break*** bewirkt den Abbruch der Schleife.

Warnung von einer Endlos-Schleife

Bei der Formulierung der Schleifen-Eintrittsbedingung ist darauf zu achten, dass die Schleife auch beendet wird. Liefert der Bedingungsausdruck nicht irgendwann das Ergebnis False, liegt eine Endlos-Schleife vor.

Beispiel.

```
while buchwert  >= restwert:
    ausgabe(jahr, abschreibung, buchwert)
    abschreibung = runden(buchwert*abschreibungssatz/100)
    buchwert = buchwert + abschreibung
```

Hier liegt ein Programmierfehler vor, der zu einer Endlos-Schleife führt. Worin besteht der Fehler?

5.1.4 Zusammenfassung

- Eine Schleife besteht aus einer oder mehreren Anweisungen (Anweisungsblock), die wiederholt auszuführen sind.
- Alle Anweisungen, die zum Schleifenkörper gehören, müssen auf derselben eingerückten Fluchtlinie beginnen.
- Die *Schleife* wird solange durchlaufen, bis die while-Bedingung den Wert False hat.
- Die Variablen in dem Bedingungsausdruck müssen aufgabenbezogene sinnvolle Werte aufweisen, da sonst die Gefahr der Endlos-Schleife besteht oder die Anweisungen des Schleifenkörpers nicht bearbeitet werden.
- Eine while-Schleife kann auch durch eine while True – if -continue – break gesteuert werden.

5.1.5 Aufgaben

Aufgabe 1

In dem Programm Zinsberechnung bei Termingeld (s. Aufgabe 2, Kapitel 3.6) sind folgende Bedingungen zu berücksichtigen:

Die Einlage muss mindestens 50.000,00 Euro betragen.

Als Laufzeit kann 90 oder 60 Tage vereinbart werden.

Der Zinssatz beträgt bei einer 90-tätigen Laufzeit 3 %, andernfalls 2 %.

Aufgabe 2

Die Kraftstoffpreise an den Tankstellen variieren in erheblichem Maße. Für die Kalkulation wird deshalb ein Durchschnittspreis benötigt.

Eingaben: getankte Liter, Rechnungsbetrag

Ausgabe: Durchschnittspreis

Wird bei der Eingabe der Literanzahl ein 0 eingegeben, soll die Eingabeschleife beendet und der Durchschnittspreis berechnet werden.

Aufgabe 3

Fortsetzung von Aufgabe 2: Die Eingaben für Liter und Rechnungsbetrag dürfen nicht negativ sein. Die Eingaben sind entsprechend abzusichern.

Aufgabe 4

Kapitalaufbau bei jährlicher Einzahlung:

Eingabe: Sparbetrag, Zinssatz, Sparziel

Erwartet wird folgende Ausgabe:

```
jährliche Einzahlung: 1100
Zinssatz: 5
Sparziel: 6000
Jahr     Zinsen      Jahresendwert
--------------------------------
 1         55.00       1155.00
 2        112.75       2367.75
 3        173.39       3641.14
 4        237.06       4978.19
 5        303.91       6382.10
Sparziel von 6000.00 erreicht
```

Aufgabe 5

Alex ist ganz begeistert von der While-True-if-Konstruktion. Er hat diese Konstruktion auf das Programm Degressive Abschreibung angewendet. Das Programm funktioniert aber nicht richtig.

```
# Wiederholungsanweisung - Degressive Abschreibung
#0805_515A5
def runden(zahl):
    return round(zahl,2)
def ausgabe(j, a, b):
    print(format(j,'2.0f')+format(a, '14.2f')+ format(b, '14.2f'))

anschaffungswert = float(input("Anschaffungswert: "))
abschreibungssatz = float(input("Abschreibungssatz: "))
restwert = float(input("Restwert: "))
buchwert = anschaffungswert
jahr = 1
while True:
    abschreibung = runden(buchwert*abschreibungssatz/100)
    buchwert = runden(buchwert - abschreibung)
    print ("Jahr  Abschreibung   Buchwert")
    print ("-"*30)
    jahr += 1
    ausgabe(jahr, abschreibung, buchwert)
    if buchwert >= restwert:
        continue
    else:
        break
```

```
ausgabe(jahr, buchwert + abschreibung -restwert, restwert)
```

Aufgabe 6

Die Absicherung gegen Fehleingaben wichtig. Bei den weiteren Eingaben von Abschreibungsprozentsatz und Restwert sind ebenfalls Grenzwerte zu beachten. Der Abschreibungsprozentsatz muss mindestens 15 sein. Der Restwert muss mindestens 800,00 Euro betragen.

Alissa hat folgende Eingaberoutine geschrieben. Testen Sie die Eingaberoutine und sichern Sie das Programm gegen Fehleingaben bei Abschreibungsprozentsatz und Restwert ab.

```
# Wiederholungsanweisung - Degressive Abschreibung
#0805_515A6
def runden(zahl):
    return round(zahl,2)

def ausgabe(j, a, b):
    print(format(j,'2.0f')+format(a, '14.2f')+ format(b, '14.2f'))

def eingabe(bedingung,anweisungstext, fehlertext):
    while True:
        ein = float(input(anweisungstext))
        if ein < bedingung:
            print(fehlertext)
            continue
        else:
            break
    return ein


anschaffungswert = eingabe(0,"Anschaffungswert: ", "Anschaffungswert kann nicht
negativ sein")
abschreibungssatz = float(input("Abschreibungssatz: "))
restwert = float(input("Restwert: "))
jahr = 1
abschreibung = runden(anschaffungswert*abschreibungssatz/100)
buchwert = runden(anschaffungswert - abschreibung)
print ("Jahr  Abschreibung   Buchwert")
print("-"*30)
while buchwert  >= restwert:
    ausgabe(jahr, abschreibung, buchwert)
    abschreibung = runden(buchwert*abschreibungssatz/100)
```

```
    buchwert = buchwert - abschreibung
    jahr += 1
ausgabe(jahr, buchwert + abschreibung -restwert, restwert)
```

5.2 for-Schleife

Die for-Anweisung kann auf sog. iterierbare Objekte angewendet werden. Iterieren kommt aus dem Lateinischen und bedeutet wiederholen. Iterierbare Objekte sind also Objekte, auf die eine Schleifenkonstruktion angewendet werden kann. Zu dieser Art von Objekten, die mit einer Iterator-Methode ausgestattet sind, zählen Zeichenkette (String), Liste und range. range kann man mit Bereich übersetzen, in dem sich abzählbare Angaben befinden. Eine Liste ist eine Zusammenstellung von Werten. Dieser Datentyp wird im nächsten Kapitel behandelt.

5.2.1 Zählschleife

Beispiel:

Beispiel 1	Beispiel 2	Beispiel 3
for y in range(5): print(y)	for y in range(1,5): print(y)	for y in range(0,5,2): print(y)
Ausgabe	Ausgabe	Ausgabe
0 1 2 3 4	1 2 3 4	0 2 4

Die Zählweise bei range beginnt mit Null, ist also nullbasiert.

Im Beispiel 1 wird y von 0 bis 4 hochgezählt.

Im Beispiel 2 beginnt die Zählung mit 1 und endet ebenfalls mit 4. Die obere Grenze, hier 5, wird nicht erreicht. Mathematisch liegt hier ein Intervall der Form [1, 5[vor.

Im dritten Beispiel wird durch den dritten Parameter die Schrittweite angegeben.

Die drei möglichen Formen von range:

1. range(endet vor): Beginnt automatisch mit 0
2. range(beginnt mit, endet vor)

3. range(beginnt mit, endet vor, schrittweite)

Beispiel

Es ist eine Tabelle zu erstellen, die den Kapitelaufbau aufzeigt. Eingegeben werden der Anlagebetrag sowie die Anlagedauer. Der Zinssatz sei 3 %.

```
Anlagebetrag: 10000
Anlagedauer (Jahre): 5
Jahr      Zinsen   Jahresendbetrag
---------------------------------
 1        300.00      10300.00
 2        318.00      10618.00
 3        328.08      10946.08
 4        338.22      11284.30
 5        348.68      11632.98
```

Programm

```
#Wiederholungsanweisung - for- range
#0805_521
1.  def runden(zahl):
        return round(zahl,2)

2.  def ausgabe(j, a, b):
3.  print(format(j,'2.0f')+format(a, '14.2f')+ format(b, '14.2f'))

4.  anlagebetrag = float(input("Anlagebetrag: "))
5.  anlagedauer = int(input("Anlagedauer (Jahre): "))
6.  zinsen = 0
7.  print("Jahr      Zinsen   Jahresendbetrag")
8.  print("---------------------------------")
9.  for n in range(1,anlagedauer+1):
10.     zinsen= runden((anlagebetrag + zinsen) * 0.03)
11.     anlagebetrag = anlagebetrag + zinsen
12.     ausgabe(n, zinsen, anlagebetrag)
```

Erläuterung

Zeilen 6 und 10: In der for-Schleife werden die Zinsen berechnet auf der Grundlage von *anlagebetrag* plus *zinsen*. Bei den Zinsen handelt es sich um einen Betrag, der im Vorjahr errechnet worden ist. Die Zinsen des Jahres 1

werden am Ende des Jahres kapitalisiert. Bei der Berechnung der Zinsen für das Jahr 2 sind sie im Kapital enthalten.

Bezogen auf das Jahr 1 sind keine Vorjahreszinsen zu berücksichtigen. Deshalb wurde in Zeile 6 die Variable *zinsen* auf Null gesetzt.

Zeile 11: Am Ende des Jahres werden die Zinsen dem Anlagebetrag zugeschlagen und ergeben so den Anlagebetrag für das nächste Jahr.

Zeile 9: Die range besteht, wenn als *anlagedauer* eine 5 eingegeben wurde, aus den Zahlen: 1, 2, 3, 4. Da die Schleife aber 5-mal durchlaufen werden muss, ist die Erweiterung um 1 erforderlich.

Die Variable *n* kann man als Laufvariable bezeichnen. Sie durchläuft die Werte von 1 bis 5.

5.2.2 for -in–Anwendung auf ein String-Objekt

Beispiel

Die Buchstaben des Wortes Python sollen untereinander ausgegeben werden.

```
# for auf Buchstaben
#0805_521_abc
programSprache ="Python"
for z in (programSprache):
    print(z)
```

Ausgabe

```
P
y
t
h
o
n
```

Die Variable *programSprache* enthält eine Folge von Zeichen bzw. einen String. Eine Stringvariable verfügt in Python über eine Iterationsfunktion. Mittels *for* wird ein Zeiger benannt, der auf die Stringvariable zeigt. Bei der Ausführung der for-Anweisung zeigt er der Reihe nach auf die Elemente des Strings. Man braucht somit weder den Anfang oder das Ende des Schleifendurchlaufs bzw. der Iteration zu programmieren.

Problem

Die Namen der Gäste sind in einer Zeichenkette erfasst worden:
„Herr Geisthaus Recklinghausen“
Für den Ausdruck auf dem Namensschild wird folgendes Format benötigt.

```
Herr
Geisthaus
Recklinghausen
```

Dieses Problem kann auf verschiedenen Wegen gelöst werden.

Um die Namensangaben in der gewünschten Form ausdrucken zu können, muss das Steuerzeichen “\n“ in die Zeichenkette eingebaut werden. “\n“ gehört zu den sogenannten Escape-Sequenzen, mit denen Druckausgaben gesteuert werden. “\n“ steht für „neue Zeile“.

Lösung 1

```
# 0805_522_1
gast = "Herr Geisthaus Recklinghausen"
neugast = ""
for zeichen in gast:
    if zeichen == " ":
        neugast = neugast+"\n"
    else:
        neugast += zeichen
print(neugast)
```

Erläuterung

In der for-Anweisung wird die Zeichenkette *gast* Buchstabe für Buchstabe darauf geprüft, ob dort ein Leerzeichen vorliegt. Falls ja, wird in die Zeichenkette *neugast* die Escape-Sequenz „***\n***“ eingetragen. Anderenfalls wird der gerade im Zugriff befindliche Buchstabe in *neugast* angefügt.

Schließlich enthält die Variable *neugast* die Zeichenkette

"Herr\nGeisthaus\nRecklinghausen"

Eine Variable vom Typ String ist ein Objekt. Ein String-Objekt kann nicht nur eine Zeichenfolge speichern, sondern verfügt auch über Methoden wie den in diesem Beispiel verwendeten Iterator.

Es wird eine neue Zeichenkette aufgebaut, da ein Hineinschreiben in eine bestehende Zeichenkette nicht möglich ist.

Escape-Sequenz

Eine weitere Escape-Sequenz ist \t. t steht für Tabulator. In der print-Anweisung wird der darauf folgende Ausdruck um eine Tabulatorposition nach rechts versetzt ausgegeben.

Würde statt “\n“ die Escape-Sequenz “\t“ verwendet, würde die Ausgabe lauten:

Herr Geisthaus Recklinghausen

Das Problem hätte auch noch auf andere Weise gelöst werden können. Es ist möglich, jede Stelle eines Strings abzufragen, ob es ein Leerzeichen enthält. Wenn ein Leerzeichen gefunden wird, erfolgt eine Zeilenschaltung, ansonsten wird das Zeichen in eine Druckschlange angefügt.

Lösung 2

```
#for-range Anwendung mit []
#0805_522_2
gast = "Herr Gasthaus Recklinghausen"
textlaenge = len(gast)
for z in range(textlaenge):
    if gast[z]== " ":
        print()                                  #Ausdruck der Druckschlange
    else:
        print(gast[z], end="")                   # Aufbau der Druckschlange
```

In *textlaenge* wird die Länge des Textes gespeichert. Die Funktion len() liefert die Länge einer Zeichenkette.
In der for-z-Schleife wird gefragt, ob die Stelle im String *gast*, auf die z zeigt, ein Leerzeichen enthält. Wenn das nicht der Fall ist, wird der Buchstabe in die Druckschlange angefügt. Der Ausdruck erfolgt nicht aufgrund des Zusatzes end=””.

Der Druckvorgang wird erst gestartet, wenn in der if-Anweisung ein Leerzeichen festgestellt wurde.

Die for-Anweisung kann hier eingesetzt werden, weil ein String iterierbar ist und über den []-Operator (Index-Operator) verfügt. Der Index in den eckigen Klammern zeigt auf die entsprechende Stelle im String. *gast*[0] adressiert das „H“. Wenn z den Wert 1 hat, zeigt gast[z] auf das „e“.

Die Technik, die in der Lösung 1 und Lösung 2 angewendet wird, kann man kombinieren.

Lösung 3

```
for buchstabe in gast:
    if buchstabe == " ":
        print()
    else:
        print(buchstabe ,end="")
```

Die Laufvariable *buchstabe* nimmt sich Buchstabe für Buchstabe der Zeichenkette vor und prüft, ob ein Leerzeichen vorliegt. Ist das der Fall, wird die Druckschlange ausgedruckt. Ansonsten wird der Buchstabe in die Druckschlange angefügt.

break

Wie die while-Schleife kann auch die for-Schleife durch **break** abgebrochen werden.

5.2.3 Zusammenfassung

- Ein Objekt ist iterierbar, wenn es über eine Iterator-Methode verfügt.
- String (Zeichenkette), Liste und range sind iterierbar.
- Eine Zählschleife wird mittels for var in range (anfangswert, endwert-1) bzw. for var in range (anfangswert, endwert-1,schrittweite) formuliert.
- range ist nullbasiert.
- for z in Stringvariable greift auf jedes Zeichen in der Stringvariablen zu.
- Die Escape-Sequenz “\n“ bewirkt eine Zeilenschaltung, “\t“ einen Tabulatorsprung.
- print(“text“,end=““): der Zusatz *end=““* verhindert die unmittelbare Ausgabe auf den Bildschirm. Erst eine print()-Anweisung ohne end=““ – Zusatz startet den Ausdruck.

5.2.4 Aufgaben

Aufgabe 1

Die derzeitige Steuergesetzgebung lässt nur die lineare Abschreibung zu.

Eingaben:

- Anschaffungswert,
- Restwert,
- Nutzungsdauer

Erwartete Ausgabe:

```
Anschaffungsbetrag: 15000
Restwert: 1000
Nutzungsdauer in Jahren: 7
Jahr   Abschreibung   Buchwert
-----------------------------
 1       2000.00      13000.00
 2       2000.00      11000.00
 3       2000.00       9000.00
 4       2000.00       7000.00
 5       2000.00       5000.00
 6       2000.00       3000.00
 7       2000.00       1000.00
```

Aufgabe 2

Gespräch am Mittagstisch. Alissa erzählt, dass sie seit ihrem 18. Lebensjahr jedes Jahr 2400,00 Euro in einen Sparplan einzahlt, der ihr eine Verzinsung von 2 % einbringt. Sie will in 40 Jahren ein Vermögen haben von etwa 150.000,00 Euro.

Alexander: „Legt das Geld in ETF an, das bringt mindestens 3 % pro Jahr. Dann hast Du dein Sparziel viel früher erreicht.

Miriam hält dagegen: „Ich habe gelesen, dass die Anlage in Aktien im Durchschnitt 4 % Rendite erbringen.

Alexander: „Dann lass uns das mal ausrechnen, wie hoch das Kapital ist nach 20, 25, 30, 35 und 40 Jahren bei den verschiedenen Zinssätzen ist.

Die Ausgabe soll in Form einer Tabelle erfolgen:

```
Jährliche Einzahlung am Jahresende? :2400
Jahr      2 %          3 %           4 %
-----------------------------------------
  20     58313.69     64488.90     71467.39
  25     76872.72     87502.23     99950.18
  30     97363.39    114181.00    134603.85
  35    119986.75    145109.00    176765.34
  40    144964.76    180963.02    228061.24
```

Aufgabe 3

Das Wort „Python“ soll im Sperrdruck ausgegeben werden.

Ausgabe: P y t h o n

Aufgabe 4

Analysieren Sie das nachstehende Programm:

```
# 0805_524A4
name = "Naßmacher"
neuname = ""
laenge = len(name)
for z in range(laenge):
    if name[z] == "ß":
        neuname = neuname+"ss"
    else:
```

```
        neuname += name[z]
print(neuname)
```

a) Was leistet die Funktion len()

b) Mit welchem Operator wird ein Zeichen an eine Zeichenkette angehängt?

c) Welchen Wert weist die Variable *neuname* auf, wenn in *name* Staßfurth steht.

Aufgabe 5

Die Ortsangaben, die einen Umlaut enthalten, machen Probleme. Es wird ein Programm benötigt, dass die Buchstaben ä, ö und ü in ae, oe und ue umwandelt.

Beispiel

Nürnberg → Nuernberg

6 Datentypen

In einem Unternehmen fallen sehr viele unterschiedliche Daten an. Adresslisten, Produktbeschreibungen, Buchungsdaten, Bestelllisten, Lohnabrechnungen, um nur einige zu nennen.

Im Sinne der Wirtschaftlichkeit müssen die Daten auf möglichst effiziente Weise gespeichert und verarbeitet werden können. Python stellt mehrere Datentypen zur Verfügung: String (Zeichenkette), Liste und Tupel, Dictionary und Mengen.

6.1 Sequentielle Datentypen

6.1.1 String - Zeichenkette

Eine Zeichenkette nimmt eine beliebige Folge von Zeichen auf. Die Zeichenfolge steht zwischen zwei Anführungszeichen oder Hochkommata.

zk = ‘Dies ist eine Zeichenkette 1 2 3 47 ‘

adresse = “Herrn Jos Wilm, 45471 Riem“

Schematische Darstellung im Arbeitsspeicher:

H	e	r	r		J	o	s		W	i	l	m	,	4	5	4	7	1		R	i	e	m
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	1	20	21	22	23

Eine Zeichenkette kann nicht verändert werden. Sollte zum Beispiel in der Zeichenkette *adresse* der Name Jos in Josef geändert werden, geht das nur durch Generierung einer neuen Zeichenkette. Das wurde im Abschnitt 5 schon behandelt.

Ein Zeichen suchen - find()

Wurde bei der Eingabe ein Komma zwischen dem Namen und der Ortsangabe gesetzt? Die Antwort bringt die Anweisung

```
#0805_611
 String untersuchen
adresse = 'Herr Jos Wilm,45781 Riem'
laenge = len(adresse)
print('Textlänge: '+ str(laenge))
#Suche
beistrich = adresse.find(',')
if beistrich >0:
   print('Komma an der Position: '+ str(beistrich))
```

```
else:
    print('Der Test enthält kein Komma')
```

Ausgabe
Textlänge: 24
Komma an der Position: 13

In der Variablen *Beistrich* wird die Position des Kommas gespeichert, sofern ein Komma vorkommt.

Die Methode find sucht im String nach dem oder den Zeichen, die in der runden Klammer angegeben sind, und liefert die Position.

Prüfung auf Vorhandensein eines Zeichens oder Zeichenfolge - in

```
ortsname = "Langstadt"
print("an" in ortsname)        → True
print ("ab" in ortsname)       → . False
print ("ab" not in ortsname) → .True
```

Mit Hilfe der Operatoren **in** bzw. **not in** kann man feststellen, ob ein bestimmtes Zeichen oder eine bestimmte Zeichenfolge in einer Zeichenkette vorkommen. Das Ergebnis ist entweder True oder False.

Positionsermittlung – index()

```
ortsname.index("s")   --→ 4
```

Der Buchstabe "s" steht an der Stelle 4.

Anmerkung: Hier sieht man, dass Python objektorientiert arbeitet. Die "Variable" *ortsname* ist ein Objekt, das über die Methode *index* verfügt.

Aufspalten einer Zeichenkette – split()

```
ortsname = "54201 Langstadt"
print(ortsname.split())          → ["54201". "Langstadt"]
print(ortsname.split("g")       → ["54201 Lan","stadt"]
```

Eine Zeichenkette wird in seine Bestandteile zerlegt und in einer Liste bereitgestellt. Der Datentyp *Liste* wird weiter unten behandelt.

Ist bei der Methode split() kein Argument angegeben, wird die Zeichenkette an der ersten Leerstelle getrennt. Ansonsten erfolgt die Auftrennung an der Stelle, an der sich der als Argument angegebene Buchstabe befindet.

Beispiel

Bei der Erfassung der Kundendaten hat man die Anrede und den Namen in eine Zeichenkette geschrieben. Für die Adressschreibung benötigt man aber Anrede und Name getrennt.

```
#BeispielSplit
#0805_611_a
name = "Herr Nonnenkamp"
anrede = name.split()
print(anrede)
print(anrede[0])
print(anrede[1])
```

Ergebnis
['Herr', 'Nonnenkamp']
Herr
Nonnenkamp

Slicing

Soll ein Teil der Zeichenkette angezeigt werden, setzt man die Slicing-Methode ein.

```
print(adresse[5:13])
```

In eckigen Klammern wird der „von:bis ausschließlich“-Bereich angegeben. Hier würde der Vorname und Name angezeigt.

Soll die Anrede ausgegeben werden, lautet die Anweisung:

```
print( :4])
```

Wird vor dem Doppelpunkt keine Angabe gemacht, beginnt der Zugriff auf der Position 0.

Interessiert nur die Adresse, lautet die Anweisung:

```
print(14:])
```

Folgt hinter dem Doppelpunkt keine Zahl, wird die Zeichenkette bis zum Ende adressiert

Count()

Bei der Ausstellung von Besucherausweisen ist zu berücksichtigen, dass die Besucher nicht nur einen, sondern möglicherweise zwei Vornamen führen.

Beispiel
Herr Jos Wilm
Herr Bernd Thomas Meier

Die Anzahl der Leerstellen, die nach der Anrede folgen, kann man als Hinweis nehmen. Bei zwei Leerstellen gibt es zwei Vornamen , zum Beispiel

Bernd Thomas Meier

Um die Anzahl eines bestimmten Zeichens oder einer Zeichenfolge festzustellen, ist die Methode count() anzuwenden.

Beispiel

```
doppelname = adresse[5:13].count(' ')
if doppelname >1:
   print('Kunde hat einen doppelten Vornamen')
else:
   print('Kunde hat nur einen Vornamen')
```

Es kann der gesamte String oder aber auch nur ein Teilstring mittels count() durchsucht werden.

6.1.2 Liste

Eine Variable des Datentyps Liste kann beliebige Daten aufnehmen.

Beispiele

```
kunde = ["Adam", 5600, "Berta", 4100, "Doro", 5000, "Cernic", 2350]
```

Die Liste *kunde* enthält die Namen der Kunden sowie die Angabe der gelieferten Heizölmenge. Die Listen enthält sowohl numerische wie auch Zeichen und Zeichenketten.

Kennzeichen einer Liste

- Die Elemente der Liste stehen in eckigen Klammern, durch Komma getrennt.
- In einer Liste können Daten unterschiedlichen Typs „aufgelistet“ werden.
- Die Liste kann erweitert oder gekürzt werden, je nach Problemstellung.
- Der Zugriff auf die Elemente einer Liste erfolgt über den Index-Operator [ ].

Das nachstehende Programm zeigt den Aufbau einer Liste und den Zugriff auf die einzelnen Elemente. Hier werden die Kundennamen und die gelieferten Heizölmengen eingegeben und an eine Liste angehängt. Nach Beendigung der Eingabe werden die Elemente der Liste ausgegeben.

```
# Listenbearbeitung
# 0805_612_1
kundeBest = []                                  # 1: Vereinbarung der Variablen
                                                #     kundeBest als Liste
weiter = "j"                                   #Vorbesetzung Bedingungsvariable
while weiter =="j":                             # Eingabe-Schleife
    name = input("Name: ")
    menge= int(input("Anzahl Liter: "))
    kundeBest.append(name)                      # 2: Anfügen an Liste kundeBest
    kundeBest.append(menge)                     #    Anfügen an Liste kundeBest
    weiter = input("weitere Eingaben? (j/n):")  #    Abfrage auf Ende der Eingabe

print(kundeBest)                                # 3: Anzeige der Liste kundeBest

for i in kundeBest:                             # 4: Anzeige der einzelnen
    print(i)                                    #    Elemente untereinander

laenge = len(kundeBest)                         # 5: Länge (= Anzahl) der Elemente
n = 0
while n < laenge:
    print(kundeBest[n], kundeBest[n+1])         # 6: Auflistung pro Kunde
    n +=2

summeLiter= 0
for z in range(1,laenge,2):                     # 7: Summierung der Liefermenge
    summeLiter += kundeBest[z]
print("Summe Liter: " + str(summeLiter))
```

Erläuterung

1 kundeBest = [] legt eine leere Liste an.

2 Die Methode append() hängt ein Element an die Liste an.

3 Der Inhalt der Liste wird ausgedruckt. Die einzelnen Elemente sind durch Komma getrennt.

['Adam', 5600, 'Berta', 4100, 'Doro', 5000, 'Cernic', 2350]

4 Der Zugriff auf einzelne Elemente erfolgt über den Index-Operator []. KundeBest[2] adressiert das 3. Element. Mit dieser Schleifenkonstruktion werden alle Elemente untereinander ausgegeben.

5 Die Funktion len(kundeBest) ermittelt, aus wie vielen Elementen die Liste besteht.

6 Mittels einer while-Schleife wird die Liste durchlaufen.

Zwei benachbarte Elemente kund[n] und kundeBest[n+1] werden ausgedruckt, zum Beispiel
Adam 5600
Berta 4100
Doro 5000
Cernic 2350

In der while-Schleife werden im ersten Durchlauf die Daten, die auf den Positionen kundeBest[0] und kundeBest [1] stehen, ausgegeben. Anschließend wird die Indexvariable *n* um 2 erhöht. Weil der Inhalt von *n* kleiner ist als der Inhalt von *laenge*, werden nun die Daten, die auf den Positionen kundeBest [2] und kundeBest[3] stehen, angezeigt. Die Schleife endet, wenn der Wert in *n* nicht mehr kleiner ist als der Wert in *laenge*.

7 Die Schleifensteuerung kann natürlich auch mittels der for-Anweisung erfolgen.

In der for-Schleife wird die Summe der Liefermengen ermittelt. Der erste Zugriff auf Liste geschieht durch kundeBest [1]. Deshalb muss in range die 1 als untere Grenze gesetzt werden. Die obere Grenze ist durch *laenge* bestimmt. Da die Mengenangaben auf den Positionen 1,3, 5 usw. stehen, ist die Schrittweite 2.

Weitere Operationen auf Listen

Listenobjekten verfügen über eine große Anzahl von Methoden. Einige seien davon hier vorgestellt. Grundlage ist das nachstehende Programm:

```
# Listenoperationen II
# 0805_612_1b
kdAdresse = ['Adam', 'Aueweg 2', '37554 Astadt', 3,
             'Berta', 'Uhuweg 9','49521 Bdorf', 2,
             'Doro','Kebach8', '52172 Konz',1,
             'Cernic', 'Ulmenweg 9','49521 Cedorf',1]

kundeBest = ["Adam", 5600, "Berta", 4100, "Doro", 5000, "Cernic", 2350]
kundeBest.extend(["Wilma", 6500])          # Anhaengen einer anderen Liste
print(kundeBest)
kundeBest[2:4]= ["Bertone", 1700]          # Position ersetzen
```

```
#print(kunde)
kundeBest.pop(3)                    # Loeschen an der 4. Stelle
kundeBest.insert(3, 4100)           # Einfügen an der 4. Stelle
kundeBest.append(['Wilmes',7500])
print(kundeBest)
if 'Wilmes'in kundeBest:            # Prüfen, ob ein bestimmtes
   print(enthalten)                 # Elemente in der Liste existiert
else:
   print('nicht bekannt')
```

Anhängen

An anderer Stelle ist auch eine Liste erstellt worden mit Kundennamen und Liefermenge. Um diese Liste ist *kundeBest* zu erweitern. Dazu wird die Methode extend() eingesetzt.

```
kundeBest = ["Adam", 5600,"Berta",4100, "Doro",5000,"Cernic",2350]
kundeBest.extend(["Wilma",6500])
```

Ergebnis

```
['Adam', 5600, 'Berta', 4100, 'Doro', 5000, 'Cernic', 2350, 'Wilma', 6500]
```

Ersetzen

Da ist einiges falsch gelaufen. Nicht Berta, sondern Bertone hat Heizöl bekommen, 1700 Liter. Der Eintrag Berta, 4100 muss ersetzt werden.

```
kunde[2:4]= ["Bertone",1700]              # Position ersetzen
```

Ergebnis

```
['Adam', 5600, 'Bertone', 1700, 'Doro', 5000, 'Cernic', 2350, 'Wilma', 6500]
```

Berta steht auf Position 3. Da Listen nullbasiert sind, hat diese Position den Index 2. Bis zur Position 4 ausschließlich soll der bisherige Eintrag ersetzt werden. Deshalb lautet die Adressierung kunde[2:4].

Löschen

Ein einzelner Eintrag muss gelöscht werden. Dazu steht die Methode pop() zur Verfügung. Die Anweisung

```
kunde.pop(3)
```

führt zu folgendem Ergebnis:

```
['Adam', 5600, 'Bertone', 'Doro', 5000, 'Cernic', 2350, 'Wilma', 6500]
```

Die Mengenangabe hinter Bertone fehlt.

Einfügen

An einer bestimmten Stelle muss ein Element eingefügt werden. Das ist möglich mit Hilfe der Methode insert():

```
kunde.insert(3,4100)
```

Ergebnis

```
['Adam', 5600, 'Bertone', 4100, 'Doro', 5000, 'Cernic', 2350, 'Wilma', 6500]
```

Listen sortieren und zusammenfügen

Beispiel: In der Kundenbestellliste sind nur Name und Menge festgehalten worden. In Hinsicht auf die weitere Verarbeitung der Bestelldaten sollen diese Daten nach Namen aufsteigend sortiert werden.

Für das Sortieren stellt Python die Methode sort() zur Verfügung.

```
stadt = ['Kopenhagen', 'Berlin', 'Paris', 'Rom']
stadt.sort()   -> ['Berlin', 'Kopenhagen', 'Paris', Rom]
```

Die Methode sort() hat jedoch den Nachteil, dass sie nur dann eingesetzt werden kann, wenn alle Elemente der Liste denselben Datentyp haben. Das ist bei der Kundenbestellliste nicht der Fall.

Um die sort-Methode nutzen zu können, müssen die Namen in eine weitere Liste kopiert werden. Diese Liste wird sortiert. Schließlich werden die Bestellmengen und die sortierten Namen in einer dritten Liste zusammen geführt.

```
["Adam", 5600, "Berta", 4100, "Cernic", 2350, 'Wilma',6500,"Doro", 5000]
                                        Namen in eine Zwischenliste KdTemp kopieren
['Adam', 'Berta', 'Cernic', 'Wilma', 'Doro']
                                                                kdTemp sortieren
['Adam', 'Berta', 'Cernic', 'Doro', 'Wilma']
                                                    Bestellmengen den Namen zuordnen
['Adam', 5600, 'Berta', 4100, 'Cernic', 2350, 'Doro', 5000, 'Wilma', 6500]
```

1. Schritt: Namen aus *kundeBest* in die Liste *kdTemp* kopieren

```
kdTemp = []                                                          #0805_612_3
laenge = len(kundeBest)
for n in range(0,laenge,2):                  # Anlegen einer echten Kopie
   kdTemp.append(kundeBest[n])
```

Die Namen aus kundeBest werden mittels der append-Methode in die Liste *kdTemp* eingetragen.

2. Schritt: Sortieren der Namensliste

```
kdTemp.sort()
```

3. Schritt: Bestellmengen und Namen zusammenführen:

```
kdBestSort = []                                  # Sortierte Bestellliste:
for i in range(0,len(kdTemp)):                   # Bestellmengen hinzufügen
```

```
    for n in range(0, len(kundeBest)):
        if kdTemp[i]==kundeBest[n]:
            kdBestSort.append(kundeBest[n])          # Anhängen Namen
            kdBestSort.append(kundeBest[n + 1])      # Anhängen Bestellmenge
```

Die Übertragung der Daten geschieht in einer geschachtelten Schleifenkonstruktion. Die i-Schleife arbeitet auf *kdTemp*, die eingeschachtelte n-Schleife greift auf *kundeBest* zu. Die i-Schleife liefert den Namen, in der n-Schleife wird der Name in *kundeBest* gesucht. Sind die beiden Namen identisch, werden der Name sowie die Bestellmengen an die Liste *kdBestSort* angefügt.

Problem

Für die Organisation der Auslieferung reicht die alphabetisch sortierte Bestellliste nicht. Man braucht auch die Adressen und die Bezirksnummer.

Lösung

```
auslieferung = []
for x in range(0,len(kdBestSort),2):
    suchname = kdBestSort[x]
    m=kdAdresse.index(suchname)
    auslieferung.extend(kdAdresse[m:m + 4])
    auslieferung.append(kdBestSort[x + 1])
```

In der x-Schleife wird *kdBestSort* Name für Name durchlaufen. Der Name im aktuellen Zugriff wird in der Variablen *suchname* gespeichert. Mit diesem Namen wird in der Liste *kdAdresse* der entsprechende Name gesucht und seine Position in der Variablen *m* festgehalten. Dazu wird die Methode index() genutzt.

Im nächsten Schritt wird die Liste *auslieferung* um die Adressdaten von der Stelle m bis zur Stelle m + 4 erweitert. Abschließend wird aus *kdBestSort* die Bestellmenge angefügt.

```
['Adam', 'Aueweg 2', '37554 Astadt', 3, 5600, 'Berta', 'Uhuweg 9', '49521 Bdorf', 2, ...
```

Mit einer solchen Liste können die Mitarbeiter in der Auslieferung aber nicht arbeiten. Sie brauchen die Daten in Tabellenform.

```
Liste der anstehenden Auslieferungen
-----------------------------------------------------
['Adam', 'Aueweg 2', '37554 Astadt', 3, 5600]
['Berta', 'Uhuweg 9', '49521 Bdorf', 2, 4100]
['Cernic', 'Ulmenweg 9', '49521 Cedorf', 1, 2350]
['Doro', 'Kebach8', '52172 Konz', 1, 5000]
['Wilma', 'Asternallee 23', '49521 Cedorf', 3, 6500]
```

Der Ausdruck erfolgt in einer for-Schleife:

```
print('Liste der anstehenden Auslieferungen')
print('-'*50)
for x in range(0, len(auslieferung), 5):
    print(auslieferung[x:x+5])
```

Speichern einer Liste

Die Kundendaten sind zu speichern. Zur Speicherung von Objekten dient das Modul pickle. Mit ihm kann man Objekte, wie zum Beispiel Listen, auf die Festplatte schreiben und einlesen.

Beispiel

```
1.  import pickle
2.  kunde = ["Adam", 5600, "Berta",4100,"Cernic", 2350,"Doro",5000, "Wilma",6500]
3.  datobj=open("kundlist.dmp","wb")
4.  pickle.dump(kunde,datobj)
5.  datobj.close
6.  print("gespeichert")
7.  kd= []
8.  datobj = open("kundlist.dmp","rb")
9.  kd= pickle.load(datobj)
10. datobj.close()
11. print(kd)
```

Erläuterung

1 Die Anweisungen für das Schreiben und Lesen von Dateien auf einer Festplatte stellt das Modul **pickle** zur Verfügung.

3 Mit der Funktion open wird ein Dateiobjekt *datobj* geöffnet im binären Schreibmodus **wb** (write binary). Der Dateiname ist kundlist.dmp.

4 Die Methode dump() lädt das Listenobjekt *kunde* in das Dateiobjekt *datobj*.

5 Das Dateiobjekt *datobj* schließt, die Daten sind auf die Festplatte übertragen.

7 Das Dateiobjekt *datobj* wird zum binären Lesen geöffnet, Dazu lautet das Argument **rb** (read binary)**.**

8 Die Methode load lädt die Daten in das Listenobjekt *kd*.

9 Das Dateiobjekt schließt. Die Daten stehen in dem Listenobjekt *kd* zur Verfügung.

Bei der Speicherung unter Anwendung der Methode pickle wird das Listenobjekt in ein Byte-Objekt umgebaut, das auch die internen Strukturen des

Objekts speichert. Dieses Byte-Objekt wird in einem Strom auf einen externen Speichern geschrieben. Die Speicherung von Objekten nennt man **Serialisieren**.

Das Einlesen des Objekts bezeichnet man dagegen als **Deserialisieren**. Dabei wird das ursprüngliche Objekt wieder hergestellt, hier unter dem Namen *kd*.

Die Argumente *wb* und *rb* stehen für write binary bzw. für read binary.

Soll die Datei nicht in dem aktuellen Arbeitsverzeichnis angelegt werden, muss der Pfad zu dem Verzeichnis angegeben werden, in dem die Datei gespeichert werden soll.

6.1.3 Tupel

Der Datentyp Tupel hat ähnliche Eigenschaften wie der Datentyp Liste. Allerdings fehlt ihm die Eigenschaft der Veränderlichkeit. In der Fachsprache nennt man das immutable. Bei einigen Problemstellungen ermöglicht er eine einfache Programmierung.

Mehrfachzuweisungen

Laufvariablen sind zu initialisieren:

m, n, i = 1, 5, 10

Diese Mehrfachzuweisung weist m den Wert 1, n den Wert 5 und i den Wert 10 zu.

Die Inhalte zweier Variablen sind zu tauschen.

x ,y = y, x

Die Variable x erhält den Wert von y und y speichert nun den Wert von x.

Packing, unpacking

Ein Tupel kann verschiedene Objekte aufnehmen. Das sei am folgenden Beispiel dargelegt, in dem 4 verschiedene Objekte in ein Tupel-Objekt gepackt werden.

```
#Tupel
0805_613
stunden= ['Walter', 75,'Susi',74,'otto',34,'Theo',26]
memo = 'Hinweis an die Mitarbeiter: Maskenpflicht!!'
tank1= 25000
tank2=30000
tup=(stunden, memo, tank1, tank2)
print(tup)
```

Ausgabe

```
(['Walter', 75, 'Susi', 74, 'otto', 34, 'Theo', 26], 'Hinweis an die Mitarbeiter: Maskenpflicht!!', 25000, 30000)
```

Zum Entpacken muss die Reihenfolge wie beim Packen beibehalten werden.

Speichern eines Tupels

In Erweiterung des Beispiels wird das Tupel auf der Festplatte gespeichert, dann wieder eingelesen und in die entsprechenden Variablen entpackt.

```
#Fortsetzung 0805_613
import pickle

datobj=open("tup.dmp","wb")
pickle.dump(tup,datobj)
datobj.close
print("gespeichert")
tip=()
datobj = open("tup.dmp","rb")
tip=pickle.load(datobj)
datobj.close()
print(tip)
stunden, memo, tank1, tank2=tip                    # Entpacken des Tupels
print(stunden)
print(memo)
print(tank1, tank2)
```

6.1.4 Zusammenfassung

- String, Liste und Tupel sind iterierbare, sequentielle Datentypen.
- Ein String besteht aus numerischen und alphanumerischen Zeichen.
- Liste und Tupel sind Container, die Daten unterschiedlichen Typs aufnehmen können.
- String und Tupel können im Gegensatz zur Liste nicht verändert werden.
- Ein String wird durch str = “”, eine Liste durch list = [] vereinbart.
- Mittels eines Indexes kann auf Elemente der Zeichenkette, Liste und Tupel zugegriffen werden.
- Die sogenannte Slicing-Methode [a:b] erlaubt den Zugriff auf einen bestimmten Bereich.
- Die Länge einer Sequenz wird mit der Funktion len() festgestellt.
- Die Methode count() ermittelt, wie oft ein Zeichen oder Zeichenkette in einem sequentiellen Datentyp vorhanden ist.

- Die Methode find() ermittelt die Position eines bestimmten Zeichens oder einer bestimmten Zeichenkette innerhalb der Sequenz.
- Eine Zeichenkette wird durch Konkatenation erweitert: s += s1
- Die Methode index() ermittelt die Position eines bestimmten Elements in einer Liste oder einer Zeichenkette.
- Innerhalb einer Liste können Elemente überschrieben wie auch gelöscht werden.
- alist.append(x) fügt x an die Liste an.
- alist.extend(blist) erweitert alist um die Elemente von blist.
- alist.insert(n,x) fügt x an der Stelle n in die Liste ein.
- Mittels alist.pop(n) wird das Element an der Stelle n aus der Liste alist entfernt.
- alist.sort() sortiert die Liste alist. Die Liste darf jedoch nur einen Datentyp beinhalten.
- Beim Tupel sind Mehrfachzuweisungen möglich.
- In einem Tupel können Daten eingepackt und später wieder ausgepackt werden.
- Das Modul pickle stellt die Methoden zur Speicherung von Objekten auf dem externen Speicher zur Verfügung.
- pickle.dump() speichert das Objekt auf einem externen Speicher (Serialisieren)
- pickle.load() lädt das Objekt vom externen Speicher in den Arbeitsspeicher (Deserialisieren).

6.1.5 Aufgaben

Aufgabe 1

Folgender Text ist gegeben: “Die Geschäftsführung wünscht allen Mitarbeiterinnen und Mitarbeitern ein schönes Wochenende.“

a) Stellen Sie fest, ob der Buchstabe „e“ häufiger in dem Text vorkommt als der Buchstabe „n“.

b) Die Umlaute ä, ö, ü durch ae, oe und ue ersetzt werden,

Aufgabe 2

Um der Gefahr zu begegnen, dass ein Unbefugter vertrauliche Texte einsehen kann, werden sie chiffriert. Beispielsweise wird aus „SCHULE“ die Buchstabenfolge „UEJWNG“.

Der zu verschlüsselnde Text wird eingegeben. Die Buchstaben des neuen Textes ergeben sich durch eine Verschiebung um 2 Stellen im Alphabet. Aus A wird C, aus B wird D, aus Y wird A usw.

Bei der Programmentwicklung können Sie davon ausgehen, dass die Eingabe aus einem Wort in Großbuchstaben ohne Umlaute und ß besteht.

Aufgabe 3

Die folgenden Aufgaben beziehen sich auf die nachstehende Tabelle:

Mitarbeiter	Geleistete Arbeitsstunden in der Woche
Viktor	39
Adamic	38
Merler	37
Dorman	36
Daniel	38
Kai	38

a) Legen Sie eine Liste *mitarbeiter* mit den obigen Angaben an:

b) Programmieren Sie eine Schleife, mit der die Daten der Liste *mitarbeiter* ausgegeben werden.

Ausgabebeispiel:

```
Adamic
38
Merler
37
```

c) Die Form der Ausgabe gefällt nicht. Sie soll in folgender Weise ausgegeben werden:

```
Adamic  38
Merler  37
```

Aufgabe 4

Das Lohnbüro benötigt einige Informationen:

a) Wie hoch ist die Summe der geleisteten Arbeitsstunden.

b) Bei welchen Mitarbeitern beträgt die Anzahl der geleisteten Wochenstunden weniger als 38?

c) Es wird eine Aufstellung der Mitarbeiter und ihre Wochenstunden in alphabetischer Reihenfolge benötigt.

d) Im Lohnbüro werden für jeden Arbeiter die geleisteten Arbeitsstunden des Tages eingegeben und zu der bereits gespeicherten Wochenstundenzahl hinzuaddiert.
Die Mitarbeiterin wünscht, dass der Name des Arbeiters angezeigt wird und sie nur noch die Stundenzahl eingeben muss.

Aufgabe 5

Speichern Sie die Daten der Mitarbeiter in einer Datei auf der Festplatte.

Aufgabe 6

Es wird ein Programm benötigt, mit dem weitere Mitarbeiter mit ihrer Wochenstundenzahl eingegeben werden können.

Laden Sie die Mitarbeiterdatei und entwickeln Sie die Eingaberoutine für die Eingabe der Daten der neuen Mitarbeiter bzw. Mitarbeiterinnen.

Die Eingabe der Wochenstundenzahl muss abgesichert werden. Die Zahl muss größer 1 und kleiner 40 sein.

Aufgabe 7

Frau Weller, die für die Organisation der Auslieferung zuständig ist, ist mit der Liste der Auslieferungen nicht zufrieden. Sie benötigt eine Liste, die nach Bezirken geordnet ist. (siehe Programm 0805_612_3)

Aufgabe 8

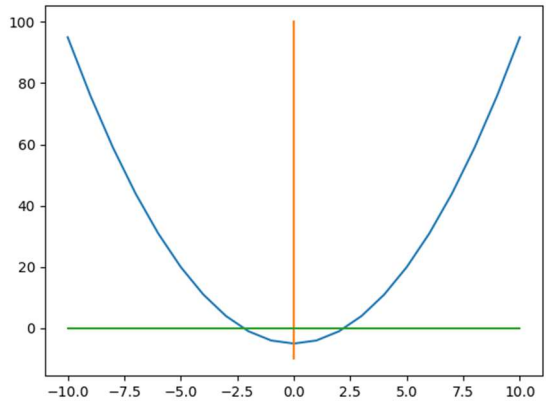
Speichern Sie die Kundenadressen und die Bestellliste auf einem externen Speicher.

6.1.6 Exkurs: Grafik

Auf der Grundlage von Listen und Tupel können auch Grafiken erzeugt werden. Dazu stellt Python das Modul matplotlib.pyplot zur Verfügung.

Beispiel

Es soll ein Graph der Funktion $y = x^2 - 5$ erzeugt werden im Bereich [-10, 10].



Der Graph wird durch das folgende Programm erzeugt:

```
import matplotlib.pyplot as plt
x = []
for n in range(-10,11,1):       #1
    x.append(n)
y = []
for i in x:                     #2
    i = i**2 - 5
    y.append(i)
plt.plot(x,y)                   #3
x1, x2 = 0, 0
y1, y2 =-10 ,100
plt.plot((x1,x2),(y1,y2))       #4
x3, x4 =-10, 10
y3 ,y4 = 0 , 0
plt.plot((x3,x4),(y3,y4))       #5
plt.show()                      #6
```

Erläuterungen

#1 Aufbau einer Liste mit einer Zahlenreihe für die x-Werte von -10 bis 10

#2 Aufbau einer Liste mit einer Zahlenreihe für die y-Werte, basierend auf den Werten der x-Zahlenreihe.

#3 plot berechnet die Anzeigepunkte für die Funktion

#4 Das Koordinatenkreuz wird aufgebaut. Bei der y-Achse haben die x-Werte jeweils den Wert 0. Die y-Werte wurden entsprechend der Funktionswerte mit -10 und 100 festgelegt. Die Methode plot berechnet die Anzeigepunkte

#5 Bei der x-Achse sind die y-Wert jeweils 0. Die x-Werte ergeben sich aus der Bereichsangabe in der Aufgabenstellung.

#6 plt.show zeigt die Kurven an.

Übung

Untersuchen Sie, was das folgende Programm leistet.

```
import matplotlib.pyplot as plt
anlagebetrag = 5000
zinssatz = 0.03
zinsen=0
x = []
for n in range(10):
    x.append(n)
y = []
for i in x:
    zinsen = round(anlagebetrag + zinsen)*zinssatz
    anlagebetrag += zinsen
    y.append(anlagebetrag)
plt.plot(x,y)
plt.bar(x,y)
plt.show()
```

6.2 Dictionary

6.2.1 Kennzeichen eines Dictionary

Übersetzt heißt Dictionary Wörterbuch. Ein Wörterbuch enthält Begriffspaare:

school: Schule;

engine:Maschine

Links vom Doppelpunkt steht der englische Begriff, rechts vom Doppelpunkt die deutsche Übersetzung. Bei diesem Datentyp werden Datenpaare gespeichert, zwischen denen eine Beziehung besteht. Das kann beispielsweise eine Geburtstagsliste oder auch eine Telefonliste sein.

Beispiel:
kdTelefon= {"Adam": "0201-528954", "Berta": "0221-741257","Doro": "0201-8569325","Wilma": "02217-8532"}

Bei einem Dictionary stehen die Datenpaare in geschweiften Klammern. Der linke Teil des Datenpaares wird Schlüssel (key) genannt. Das zugeordnete Datum ist der Wert (value):

```
{Schlüssel:Wert}
```

Der Doppelpunkt verbindet Schlüssel und Wert. Enthält ein Dictionary mehrere Wertepaare, werden sie durch Komma voneinander getrennt.

Das Dictionary ist in geschweiften Klammern eingeschlossen.

Syntax

```
dictionar = {Schlüssel1: Wert1, Schlüssel2:Wert2, ..}
```

6.2.2 Operationen auf Dictionary

Abfrage eines Wertes

Welche Telefonnummer hat Doro?

```
print(kdTelefon["Doro"])  → 0201-8569325
```

Der Zugriff erfolgt über den Schlüssel (key).

Syntax

```
dictionar[schluessel]  →  liefert den Wert zum angegebenen Schlüssel
```

Abfrage auf einen Schlüssel

Man sucht zu einem Schlüssel den Wert. Es könnte aber sein, dass der Schlüssel nicht existiert.

Lösung

```
print(kdTelefon.get("Berti","nicht vorhanden“)
```

In dem obigen Dictionary gibt es keinen Schlüssel Berti. Deshalb wird hier *nicht vorhanden* ausgegeben.

Dictionary ausgeben

Die Methode items() liefert eine Übersicht über das Dictionary.

```
print(kdTelefon.items())
```

Ergebnis

```
dict_items([('Adam', '0201-528954'), ('Berta', '0221-741257'), ('Doro', '0201-8569325'), ('Wilma', '02217-8532')])
```

Anzeige aller Schlüssel

Man möchte sich die Schlüssel anzeigen lassen.

```
print(kdTelefon.keys())
```

Ergebnis

```
dict_keys(['Adam', 'Berta', 'Doro', 'Wilma'])
```

Ein Dictionary an ein Dictionary anfügen

```
telneu = {"Cernic":"05321-5418"}
kdTelefon.update(telneu)
```

Mittels der Methode update() wird ein Dictionary an ein anderes angefügt.

Ein neues Element einfügen

Die Telefonnummer eines neuen Kunden ist per Tastatur einzugeben und in das Dictionary einzufügen.

Lösung

```
name = input("Name des Kunden: ")
telNr = input("Telefonnummer: ")
kdTelefon[name] = telNr
```

Der Inhalt von name ist ein neuer Schlüssel. Er wird zusammen mit seinem zugeordneten Wert in das Dictionary *kdTelefon* eingetragen.

Löschen eines Elements

Die Daten des zuletzt eingegebenen Kunden Simon sind zu löschen.

```
del kdTelefon["Simon"]
```

Ausgabe aller Elemente in Tabellenform

Die Namen und Telefonnummern sind in Form einer Tabelle auszugeben.

```
for schluessel in kdTelefon:
    print(schluessel, kdTelefon[schluessel])
```

Eine for-Anweisung steuert die Ausgabeschleife. Das Programm greift der Reihe nach auf die Schlüssel zu und gibt die dazu gehörenden Werte aus.

Ergebnis

```
Adam 0201-528954
Berta 0221-741257
Doro 0201-8569325
Wilma 02217-8532
Cernic 0532-5418
```

6.2.3 Kombination Dictionary, Liste und Datei

Der Wert zu einem Schlüssel kann auch aus einer Liste bestehen.

Beispiel

```
kdAdresse= {"Adam": ["Aueweg 2", "37554 Astadt", 3],
            "Berta":["Uhuweg 9", "49521 Bdorf", 2],
```

```
                "Doro":["Kebach 8", "52172 Konz",1]}
```

Der Wert zu den Schlüsseln besteht hier aus einer Liste mit den Angaben zur Adresse und Verkaufsbezirk. Soll ein neuer Kunde eingefügt werden, werden die Daten erst in eine Liste gespeichert und anschließend in das Dictionary eingefügt.

```
1.  # Dictionary
2.  # 0805_623
3.  kdAdresse= {
4.  "Adam": ["Aueweg 2", "37554 Astadt", 3],
5.  "Berta": ["Uhuweg 9", "49521 Bdorf", 2],
6.  "Doro": ["Kebach 8", "52172 Konz", 1]}
7.  # Eingabe der Kundendaten
8.  name = input("Name: ")
9.  plzOrt = input("PLZ Ort: ")
10. strasse = input("Strasse: ")
11. bezirk = int(input("Eingabe Bezirk: "))
12. # Zuweisung der Kundendaten an die Liste adr
13. adr=[]
14. adr.append(plzOrt)
15. adr.append(strasse)
16. adr.append(bezirk)
17. # Eintrag des Kundennamens und der Adressdaten in das Dictionary
18. kdAdresse[name]=adr
```

Erläuterung

In den Anweisungen der Zeilen 8 bis 11 werden die Kundendaten erfasst. In der Zeile 13 wird eine leere Liste mit dem Namen *adr* erzeugt, in die dann die eingegebenen Kundendaten angehängt werden (Zeilen 7 bis 9). Der Kundenname in der Variablen *name* wird nicht in die Liste *adr* eingeschrieben, da der Name der Schlüssel im Dictionary sein wird.

In Zeile 18 erfolgt der Eintrag des Schlüssels mit seiner zugehörigen Liste *adr* in das Dictionary *kdAdresse*.

Folgende Kundendaten wurden eingegeben.

```
Name: Simon
PLZ Ort: 49552 Cedorf
Strasse: Bachweg 9
Eingabe Bezirk: 1
```

Die Anweisung print(kdAdresse) zeigt den erweiterten Inhalt von kdAdresse an.

```
{'Adam': ['Aueweg 2', '37554 Astadt', 3], 'Berta': ['Uhuweg 9', '49521 Bdorf', 2],
'Doro': ['Kebach 8', '52172 Konz', 1], 'Simon': ['49522 Cedorf', 'Bachweg 9', 1]}
```

Speichern eines Dictionary in einer Datei

Die Speicherung eines Dictionary erfolgt in gleicher Weise wie bei einer Liste.

```
1. import pickle
2. datobj= open("kundenadresse.dmp", "wb")
3. pickle.dump(kdAdresse,datobj)
4. datobj.close
5. print("gespeichert")
6. kdadr= {}
7. datobj= open("kundenadresse.dmp", "rb")
8. kdadr= pickle.load(datobj)
9. datobj.close()
```

Für die Speicherung wird die Methode pickle benötigt. In Zeile 2 wird das Datenobjekt geöffnet zum binären Schreiben. Anschließend wird *kdAdresse* in das Datenobjekt geladen. In Zeile 4 ist die Speicherung erfolgt, das Datenobjekt wird geschlossen. Die Daten sind in der Datei *kdadresse.dmp* im Arbeitsverzeichnis auf der Festplatte gespeichert.

Vor dem Einlesen wird in Zeile 6 ein leeres Dictionary *kdadr* angelegt. In Zeile 7 wird die Datei *kundenadresse.dmp* zum binären Lesen geöffnet. In Zeile 8 werden die Daten deserialisiert und in *kdadr* geladen. Abschließend wird das Datenobjekt wieder geschlossen.

Die Daten stehen nun in dem Dictionay *kdadr* zur Verfügung.

6.2.4 Zusammenfassung

- Das Dictionary nimmt Begriffspaare auf:
  dict={key1:value1,key2:value2,… }
- dict.items() liefert die Struktur des Dictionary
- dict[key] liefert den zugehörigen Wert.
- dict.update({a:b}) fügt ein weiteres Dictionary an.
- dict[key]=value trägt ein neues Begriffspaar ein.
- Der Wert zu einem Schlüssel kann aus einer Liste bestehen.

6.2.5 Aufgaben

Aufgabe 1

Legen Sie ein Dictionary mit folgenden Daten an:

Mitarbeiter	Stundenlohn
Viktor	18,70
Kai	18,40
Adamic	19.80
Merler	20.15
Dorman	17.30
Daniel	18.40

Schreiben Sie die Anweisung, mit der die Daten in einer Übersicht angezeigt werden.

Aufgabe 2

Schreiben Sie die Anweisung, mit der die Namen der Mitarbeiter angezeigt werden.

Aufgabe 3

Geben Sie die Namen weiterer Mitarbeiter und ihren Stundenlohn ein, und fügen Sie die Daten in das Dictionary ein.

(Nehmen Sie die Namen, die Sie bei der Aufgabe 9 des Abschnitts 6.1.5 zusätzlich eingegeben haben.)

Aufgabe 4

Schreiben Sie die Ausgaberoutine, mit der die Daten der Mitarbeiter angezeigt werden.

Aufgabe 5

Speichern Sie das Dictionary.

Aufgabe 6

Das Lohnbüro braucht ein Programm, mit dem der Bruttowochenlohn der Mitarbeiter berechnet wird.

Laden Sie die beiden Mitarbeiterdateien und erstellen Sie eine Bruttolohnliste.

Ausgabebeispiel

```
Name    Wochenstunden     StdLohn      Bruttolohn
-------------------------------------------------
Adamic        38            19.80        752.40
Daniel        38            18.40        699.20
Dorman        39            17.30        674.70
Merler        37            21.15        782.55
```

….

Aufgabe 7

Das Verschlüsselungsprogramm (s. 6.1.5 Aufgabe 2) ist zu leicht zu durchschauen. Der Buchstabenaustausch soll unter Einsatz eines Dictionary umgestaltet werden: {‘A‘:‘F‘,‘B‘:‘A‘,‘C‘:‘W,‘D‘:‘D‘, ….}

Legen Sie eine Chiffriertabelle als Dictionary an und schreiben Sie das Programm zum Buchstabenaustausch.

6.3 Mengen

6.3.1 Charakteristik

Eine Menge enthält Elemente beliebigen Typs – Zeichen, Zeichenketten, Zahlen. Charakteristisch ist, dass jedes Element nur einmal vorkommt. Das zeigt das nachstehende Beispiel.

```
#0805_631_1
from random import *
zahlen = set()                  # 1
for x in range (15):
    zufall = randint(1,10)      # 2
    zahlen.add(zufall)          # 3
    print(zufall)

print(zahlen)                   # 4
```

```
6
5
10
9
3
2
7
1
6
5
6
8
10
4
9
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Nach dem Import des Zufallszahlen-Moduls wird die Menge *zahlen* vereinbart (#1). In der for-x-Schleife wird 15 Mal eine Zufallszahl zwischen 1 und

10 gebildet (#2) und der Menge *zahlen* hinzugefügt (#3). Die Zufallszahlen werden ausgegeben. Abschließend werden die Elemente von zahlen angezeigt (#4)

Es wurden mehrere gleichlautende Zahlen gebildet. In der Menge *zahlen* ist aber jede Zahl nur einmal vertreten.

6.3.2 Operationen auf Mengen

Als Operationen sind die aus der Mathematik bekannten Mengenoperationen wie Bildung von Vereinigungs- und Durchschnittsmenge zulässig. Das sei an dem nachstehenden Beispiel verdeutlicht.

Beispiel

Die Firma Klender hatte einen Tag der offenen Tür abgehalten. Die Besucher konnten in zwei Hallen die Betriebsausstattung betrachten und mit den Angestellten diskutieren. Die Geschäftsleitung wünscht zu wissen, wie viele und welche Kunden gekommen sowie welche Mitarbeiter im Einsatz waren.

In der DV-Abteilung überlegt man, wie man am einfachsten die Informationswünsche erfüllen kann. Entsprechend der Sicherheitsvorschriften sind alle Personen, die in die Hallen hineingegangen sind, in Listen erfasst worden.

Alexander: „Ein Fall für den Datentyp Menge!“

Lösung

Die Namen der Personen, die die Hallen besu cht haben, wurden am Rechner erfasst und in den Listen *halle1* bzw. *halle 2* gespeichert.

```
#0805_632
# Menge
halle1=['Meier', 'Munter', Kai, 'Bertone', 'Zylinki', 'Keller', 'Viktor']
halle2=['Munter', 'Baumann', 'Greber', 'Keller', 'Merler',‘Daniel’, 'Berta']
halle1.extend(halle2)
beideHallen=set(halle1)                                    # Bildung der Menge
print(beideHallen)
mitarbeiter = ['Adamic','Viktor','Merler','Daniel',‘Kai’,‘Dorman’]
mitarbeiterMenge=set(mitarbeiter)                          # Bildung der Menge
schnittmenge= beideHallen & mitarbeiterMenge               # Bildung Durch-
                                                           #schnittsmenge
print("Mitarbeiter im Einsatz")
print(schnittmenge)
differenzMenge = beideHallen - schnittmenge                # Bildung Differenz
```

```
                                                    # menge
print('Kunden: ')
print(differenzMenge)
n = len(differenzMenge)                          # Ermittlung Anzahl Elemente
print('Anzahl Besucher '+ str(n))
```

Erläuterungen

Zur Bildung einer Menge wird die Anweisung set() eingesetzt. Hier wird die Menge aus der Liste halle1 und halle2 gebildet und angezeigt.

Ergebnis:

```
{'Merler', 'Daniel', 'Kai', 'Baumann', 'Zylinki', 'Meier', 'Keller', 'Greber', 'Viktor',
'Bertone', 'Berta', 'Munter'}
```

Das Kennzeichen einer Menge sind geschweifte Klammern.

Anschließend wird eine weitere Menge aus der Liste der Mitarbeiter aufgebaut.

Der Operator & ermittelt die Durchschnittsmenge. In der Anweisung

```
Schnittmenge = beideHallen & mitarbeiterMenge
```

wird die Durchschnittsmenge der beiden Mengen gebildet. Die Schnittmenge enthält die Elemente, die in beiden Mengen vorkommen. Hier sind es die Namen der Mitarbeiter, die in den Hallen tätig waren.

Ergebnis:

```
Mitarbeiter im Einsatz
{'Merler', 'Kai', 'Viktor', 'Daniel'}
```

Um zu ermitteln, welche Kunden die Ausstellung besucht haben, muss die Differenzmenge von *beideHallen* und *schnittmenge* ermittelt werden.

Ergebnis

```
Kunden beim Tag der offenen Tür:
{'Baumann', 'Zylinki', 'Meier', 'Keller', 'Greber', 'Bertone', 'Berta', 'Munter'}
```

Die Anzahl der Elemente einer Menge wird mittels der Funktion len() ermittelt.

```
n = len(differenzMenge)                          # Ermittlung Anzahl Elemente
print('Anzahl Besucher '+ str(n))
```

Ergebnis: Besucher 8

Mit diesem Programm werden die Informationswünsche der Geschäftsleitung erfüllt.

6.3.3 Zusammenfassung

- set() erzeugt ein Mengenobjekt.
- Mit der Methode add() wird ein Element der Menge angefügt: menge.add(element)
- Die Durchschnittsmenge bildet der Operator &; mschnitt = m1 & m2
- Die Differenzmenge wird durch menge1 – menge2 erzeugt.
- Die Vereinigungsmenge wird durch mges = menge1 | menge2 gebildet
- Mittels menge.discard(element) wird ein Element aus der Menge entfernt.
- Zur Ausgabe der Elemente einer Menge ist eine for-Anweisung einzusetzen.
- if x in menge prüft, ob das Element x in der Menge menge enthalten ist.
- Die Funktion len(menge) liefert die Anzahl der Elemente einer Menge.

6.3.4 Aufgaben

Aufgabe 1

In der Auftragsbearbeitung gibt die Sachbearbeiterin die Fahraufträge für die Auslieferungsfahrer in eine Liste ein. Die Auslieferungsfahrer melden, welche Auslieferung sie durchgeführt haben. Zu Geschäftsschluss braucht die Geschäftsleitung die Information, ob alle Auslieferungen erfolgt bzw. welche Aufträge nicht erledigt sind.

```
#Auslieferungskontrolle
#0808_633_A1
auftrag1 = ['B12','B15','A12','A23','A43','C43','C56']
auftrag2 = ['De12','BT15','DA12','DA23','FA43','AC43','AC56']
auslieferung1 =['B12','A12','A23','A43','C56']
auslieferung2 = ['De12','BT15','DA12','DA23']
auslieferung3= ['AC43','AC56']
# Sind alle Auslieferungen erfolgt?
# Welche Aufträge sind noch nicht
```

Aufgabe 2

Die Sicherheitsbestimmungen für den Laborbereich der Firma Klender sehen vor, dass a) nur bestimmte Personen in den Laborbereich eintreten dürfen und b) dass jederzeit festgestellt werden kann, wer sich im Laborbereich aufhält.

In der IT-Abteilung hatte ein Kollege mit der Programmierung der Zu- und Abgangskontrolle begonnen.

```
#Sicherheit
#0805_633_A2
mitarbeiter = ['Adamic','Viktor','Merler','Daniel','Kai','Dorman', ,'Karl','Alexa']
# Bilde mitarbeiterMenge

laborZugelassen={'Cernic','Egon','Doro','Berta','Gewerbeaufsicht']
imlabor = {}
if zugang  :
   person = input("Eingabe Name: ")
   # Gehört die Person zu den Mitarbeitern, die zum Labor zugelassen sind?
   if
      #
   else:
      # Zurückweisung
else:
   # Entfernung aus dem Labor

# Wer ist im Labor?
```

Das Programm muss so beschaffen sein, dass die Zu- und Abgänge erfasst und auf Berechtigung kontrolliert werden.

Das Ende der Arbeitszeit wird durch eine Eingabe angezeigt wird. Es ist dann noch ist zu prüfen, ob sich jemand im Laborbereich aufhält.

Aufgabe 3

In der Mittagspause am Montag dreht sich das Gespräch um die Panne bei der Ziehung der Lottozahlen. Einer der Fahrer fragt: „Kann ein Computer nicht einfach die Zahlen ermitteln. Alexander, schreib doch mal ein Programm dafür."

Entwickeln Sie ein Programm, das genau 6 Zahlen aus 49 ermittelt.

6.4 Verarbeitung von CSV-Dateien

Python verfügt über ein Modul, mit dem CSV-Dateien eingelesen und in diesem Format auch geschrieben werden können. Die Abkürzung steht für Comma Separated Values, Werte, die durch Komma voneinander getrennt sind. Dieser Dateityp wird von EXCEL unterstützt. Damit ist es möglich, Dateien, die in EXCEL erzeugt wurden, in ein Python-Programm einzulesen, die Daten zu verarbeiten und abschließend die Daten in einer CSV-Datei zu speichern.

Beispiel

In der Firma Klender werden die Arbeitsstunden der Arbeiter in einer Exeldatei erfasst.

Name	Woche 1	Woche 2	Woche 3	Woche 4
Albert	38	40	39	38
Berta	37	39	40	41
Dora	39	40	35	38
Carlos	38	35	40	40
Elfie	39	38	39	41
Guzal	40	41	38	36
Heiner	39	40	38	35

Die Daten werden in der Datei Stundenzettel_Jan.csv gespeichert.

Zur Berechnung des Bruttolohns werden die Daten in ein Python-Programm eingelesen und verarbeitet. Das Programm addiert die Arbeitsstunden der Arbeiter und multipliziert die Summe mit dem jeweiligen Stundensatz.

```
persdat={'Albert':15.60,'Berta':16.90,'Dora':18.70,'Carlos':19.20,
        'Elfie':18.90,'Guzal':19.20,'Heiner':18.70}
```

Die Stundensätze werden in einem Dictionary bereitgestellt.

Die Monatslöhne werden abschließend in der Datei lohnJanuar.csv gespeichert und stehen für die weiteren Schritte der Lohnabrechnung bereit.

Albert	2418.0
Berta	2653.3
Dora	2842.4
Carlos	2937.6
Elfie	2967.3
Guzal	2976.0
Heiner	2842.4

Für die Bruttolohnberechnung wird das nachstehende Programm eingesetzt.

```python
#Einlesen eines Stundenzettels aus einer csv-Datei
#0805_64_BruttolohnCSV
import csv
#Stundensätze der Arbeiter
from typing import List

persdat={'Albert':15.60,'Berta':16.90,'Dora':18.70,'Carlos':19.20,'Elfie':18.90,'Guzal':19.20,'Heiner':18.70}

stdliste = []
leser=csv.reader(open("Stundenzettel_Jan.csv"), delimiter=";")
zeilennr = 0                    # Mittels der Methode csv.reader() werden die Erlä
for zeile in leser:             # Daten in die Variable leser eingelesen und stehen dort
    if zeilennr == 0:           # zeilenweise in Form einer Liste zur Verfügung
        zeilennr += 1            # die Überschriftszeile wird eingelesen, aber nicht verarbeitet
        continue
    else:
        if zeile==[]:            # für den Fall, dass die Datei eine Leerzeile enthält.
            continue
        else:
            stdliste.append(zeile[0])   # was geschieht hier?
            sum = 0
            for x in range(1,len(zeile)):
                sum += int(zeile[x])
            stdliste.append(str(sum))
            print(stdliste)
            zeilennr += 1

print("Anzahl der eingelesenen Sätze: " + str(zeilennr))
print(stdliste)
lohndatei =[]
with open('4lohnJanuar.csv','w',newline ='') as datei:    # Vereinbarun Dateiname lohnJanuar.csv
    schreiber=csv.writer(datei)                           #  Methode csv-writer bekommt als Parameter datei zugewiesen
    print ('Bruttolohn')
    print('-'*20)
    for x in range(0,len(stdliste),2):
        lohn =round(int(stdliste[x+1])*persdat[stdliste[x]],2)
        name = stdliste[x]
        print(name +'\t' + format(lohn,'8.2f'))                        # was geschieht hier?
        blohn=str(lohn)
        lohnzeile = name + ';' + blohn
        lohndatei.append(lohnzeile)
        # Schreiben in csv-Datei
        schreiber.writerow(lohndatei)                      # zeilenweise Ausgabe in die Datei
        lohndatei.pop()                                    # setzt die Lohndatei zurück auf []
```

Übungen

Analysieren Sie das das Programm. Es enthält eine Reihe von Ausgabe-Anweisungen. Daran können Sie ableiten, was die Daten verarbeitet werden.

Erläutern Sie, was in den Zeilen 18 bis 24 geschieht?

Die for-x-Schleife in den Zeilen 33 bis 42 hat es in sich. Durchdringen Sie den Code.

Wenn das Programm korrekt arbeitet, können einige print-Anweisungen gelöscht werden. Nehmen die Löschungen vor.

Aufgabe

In der Ausgabedatei *lohnJanuar.csv* fehlt die Überschrift, wie zum Beispiel Name Monatsbrutto. Für weitere Verarbeitungsschritte ist es aber wichtig, dass die Spalten mit einer Überschrift versehen sind.

Name	Monatsbrutto
Albert	2418.0
Berta	2653.3
Dora	2842.4
Carlos	2937.6
Elfie	2967.3
Guzal	2976.0
Heiner	2842.4

Nehmen Sie die entsprechenden Ergänzungen vor.

7 Objektorientierte Programmierung

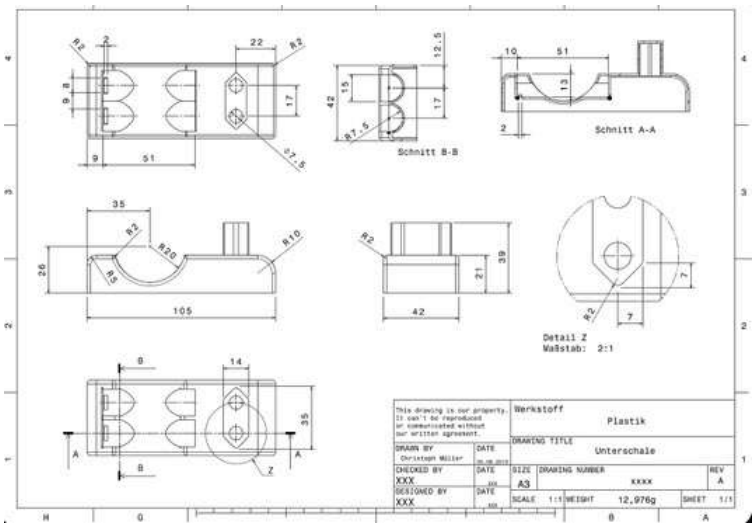
7.1 Klasse und Objekt

Die Objektorientierung ist ein Konzept, bei dem Daten und Programmfunktionen zusammenfasst werden. Dieser Programmieransatz unterscheidet sich von der prozeduralen Programmierung, die bislang behandelt worden ist.

Klasse, Objekt und Instanz sind zentrale Begriffe in der objektorientierten Programmierung. Ein Objekt ist ein Gegenstand der realen wie auch der ideellen Welt. Das können Häuser, Autos, aber auch Verträge sein.

Eine Klasse beschreibt die Eigenschaften und Fähigkeiten eines Objekts. Sie ist eine Art Bauplan für die Schaffung eines Objekts. Jedes Objekt ist eine Instanz der Klasse. Die Fähigkeiten, über die ein Objekt verfügt, nennt man Methoden.

Beispiel



Eine Konstruktionszeichnung einer Ladeschale eines Telefons stellt im Sinne der objektorientierten Programmierung eine Klasse dar. Es beschreibt das Aussehen, die Bauweise sowie die Fähigkeiten des Geräts. Das Resultat der Umsetzung der Konstruktionszeichnung ist dann das Objekt. Aufgrund der Konstruktionszeichnung können dann beliebig viele Geräte, also Objekte erzeugt werden.

7.2 Struktur und Entwicklung eines objektorientierten Programms

7.2.1 Eigenschaften und Methoden

Nicht nur Konstruktionszeichnungen, sondern auch Verträge können Klassen sein. Wenn jemand ein Sparkonto bei einem Kreditinstitut anlegen möchte, muss er einen Kontovertrag mit der Bank abschließen. Der Kontovertrag beinhaltet unter anderem folgende Bestimmungen:

- Das Sparbuch muss auf eine bestimmte Person ausgestellt sein.
- Die Bank führt das Konto auf Guthabenbasis. Kontoüberziehungen sind nicht zulässig. Das Mindestguthaben beträgt 1,00 Euro.
- Die Bank vergütet die eingezahlten Beträge mit Zinsen zu einem festgelegten Zinssatz und schreibt am Ende des Jahres die Zinsen dem Konto gut (Kapitalisierung der Zinsen).
- Der Kontoinhaber erhält dazu ein Sparbuch.

Aus den Bestimmungen des Vertrages ergibt sich, dass das Konto über folgende Attribute verfügen muss:

- Angaben zum Kontoinhaber
- Kontostand
- Aufgelaufene Zinsen (Zinsguthaben)
- Kontonummer bzw. IBAN

Damit der Kontostand und die Zinsen berechnet werden können, sind folgende Methoden erforderlich:

- Berechnung des Kontostandes nach Ein- bzw. Auszahlungen
- Berechnung der Zinsen
- Fortschreibung des Zinsguthabens durch Zu- und Abbuchen von Zinsen
- Zuschreibung der Zinsen zum Sparkapital am 31.12. eines Jahres

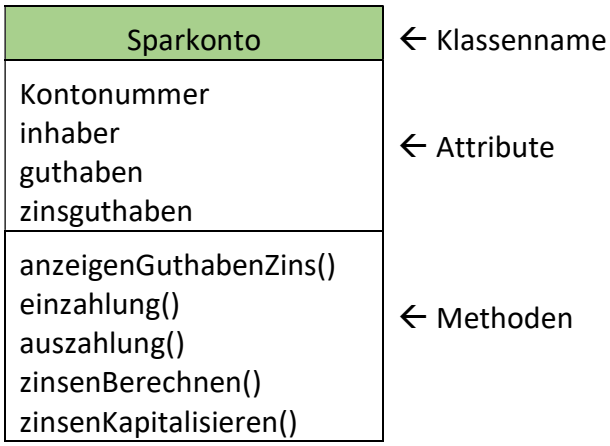
Ein Programm, mit dem Sparkonten verwaltet werden, muss über die obigen Eigenschaften und Methoden verfügen.

7.2.2. Repräsentation eines Objektes im Rechner

Im Arbeitsspeicher eines Computers kann natürlich nicht ein realer Gegenstand, wie zum Beispiel das Sparbuch, stehen. Es wird vielmehr durch eine binäre Darstellung repräsentiert. Wie ein solches Objekt aufzubauen ist, beschreibt die Klasse mit Hilfe von Anweisungen an das Computersystem.

Zur Entwicklung einer Klasse bedient man sich eines UML-Klassendiagramms. UML ist die Abkürzung von Unified Modeling Language. In einem

solchen Diagramm werden die Attribute und Methoden dargestellt.



7.2.3 Implementierung

Auf der Grundlage des Klassendiagramms wird die Klasse in einer objektorientierten Programmiersprache, hier Python, formuliert.

```
# Sparkonto                                                    #0805_723
    from datetime import

    class Sparkonto():                                                #1
        def __init__(self, kontonr, inhaber, guthaben, zinsguthaben):  #2
            self.kontonr = kontonr
            self.inhaber = inhaber
            self.guthaben = guthaben
            self.zins = zinsguthaben
            self.zinssatz = 3/100

        def zeigeGuthabenZins(self):                                  #3
            print(self.inhaber, self.guthaben, self.zins)

        def einzahlung(self, eingezahlterBetrag, datum):              #4
            self.guthaben += eingezahlterBetrag
            self.zins += self.zinsenBerechnen(eingezahlterBetrag, datum)

        def abhebung(self, auszahlung, datum):                        #5
            self.guthaben -= auszahlung
            self.zins -= self.zinsenBerechnen(auszahlung, datum)

        def zinsenBerechnen(self, betrag, datum):                     #6
            tage = ( date(2021,12,31) - datum).days
            return betrag*tage*self.zinssatz/365
```

```
# Hauptprogramm - main – Start                                    #7
    kd1 = Sparkonto(458,"Elsa" ,510.45, 12.50)
    kd2 = Sparkonto(459, "Fritz", 680.33, 13.85)
    kd1.zeigeGuthabenZins()                                       #8
    kd1.einzahlung(1000.00, date(2021,12,1))                      #9
    kd1.zeigeGuthabenZins()
    kd2.abhebung(500, date(2021,12,1))
    kd2.zeigeGuthabenZins()
```

Erläuterungen

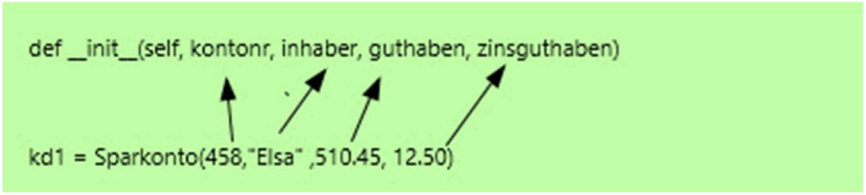
Das Programm besteht aus drei Teilen. Zuerst werden die Module bzw. Bibliotheken eingeladen, die für die folgenden Anweisungen benötigt werden. Hier wird das Modul datetime für die Tageberechnung gebraucht. Es folgt dann die Klassenbeschreibung (Zeile 2 bis Zeile 19).

Der dritte Teil ist Hauptprogramm. Vor ihm aus erfolgt die Aufforderung zu Objektbildung und der Aufruf der Methoden (Zeile 20 bis Zeile 26).

Zeile	
#1	Die Klassendefinition beginnt mit dem Schlüsselwort **class**. Es folgt der Name der Klasse. Die Zeile wird mit einem Doppelpunkt abgeschlossen. Es ist üblich, den Namen einer Klasse mit einem Großbuchstaben zu beginnen.
#2	def __init__ leitet den Konstruktor ein. Der Konstruktor hat die Aufgabe, eine Instanz der Klasse im Speicher anzulegen. Hinter def folgt ein Leerzeichen. Vor und nach init sind jeweils 2 Unterstriche einzutippen:__init__ Darauf folgt eine Parameterliste. Der erste Parameter in der Klammer lautet immer *self*. Darin speichert das System die Adresse der aktuellen Instanz bzw. des aktuellen Objekts. Anschließend werden die Parameter aufgeführt, die die Attribute der Klasse ausmachen. Sie nehmen die Daten auf, die bei der Initialisierung der Klasse übergeben werden (s. #7). Die übergebenen Daten werden in den Folgezeilen in die Klasse übernommen. Die aufnehmenden Variablen werden Instanzattribute oder Objektattribute genannt. Die Objektattribute haben stets die „Vorsilbe“ self.

	Hier können auch weitere Objektattribute definiert werden, wie zum Beispiel self.zinssatz = 3/100.
#3	Die Methode *zeigeGuthabenZins(self)* wird definiert. Das einleitende Schlüsselwort lautet ***def***. In der Parameterliste wird zuerst wieder ***self*** aufgeführt. Die Methode wird aus dem Hauptprogramm aufgerufen (s. #8). Der Inhalt der Objektattribute wird ausgegeben.
#4	In der Methode *einzahlung(self, einzahlung, datum)* wird der eingezahlte Betrag zum Sparguthaben hinzu addiert. Zur Fortschreibung des Zinsguthabens muss aber erst noch der Zinsbetrag berechnet werden. Den ermittelt und liefert die Funktion *self.zinsberechnung()*, an die der eingezahlte Betrag und das Einzahlungsdatum übergeben werden.
#5	Die Methode *abhebung()* ist in der gleichen Weise konstruiert wie die Methode *einzahlung().*

#6	An die Methode *zinsenBerechnen(self, betrag, datum)* werden der Ein- bzw. Auszahlungsbetrag sowie das Datum des Zahlungsvorgangs übermittelt. In der Folgezeile wird die Anzahl der Tage berechnet.
	`tage = ( date(2021,12,31) - datum).days`
	Bei Sparkonten werden bei jedem Zahlungsvorgang die Zinsen immer bis zum 31.12. eines Jahres berechnet und dem Zinsguthaben hinzu addiert oder von ihm abgezogen.
	Bei der Berechnung der Differenz der beiden Datumsangaben muss angegeben werden, worauf sich der Unterschied beziehen soll. Hier ist es die Anzahl der Tage. Deshalb ist die days-Methode anzugeben.
	In der return-Anweisung
	`return betrag*tage*self.zinssatz/365`
	wird der Zinsbetrag berechnet und an die aufrufende Stelle geliefert. Zu beachten ist hier, dass zinssatz die „Vorsilbe“ self hat. *zinssatz* ist hier ein Objektattribut. Es wurde deklariert mit self.zinssatz = 3/100. Die Variable *tage* kann man als lokale Variable bezeichnen. Sie „lebt“ nur innerhalb der Methode *zinsberechnung()*.

#7	Hier beginnt das Hauptprogramm. Man kann es auch als eine Art Steuerleiste bezeichnen. Von hier aus erfolgt die Anweisung zur Objektbildung.  Mit dieser Anweisung wird ein Sparkonto mit der Kontonummer 458, Inhaber Elsa, Guthaben 510,45 und Zinsguthaben 12,50 angelegt. kd1 ist die Objektvariable. Sie enthält die Speicheradresse, unter der das Objekt im Hauptspeicher abgelegt ist.
#8	Mittels kd1.zeigeGuthabenZins() wird die entsprechende Methode (s. #3) aufgerufen. Ausgabe: `Elsa 510.45 12.5`
#9	kd1.einzahlung(1000.00, date(2021,12,1)) ruft die entsprechende Methode auf (s. #4) und übergibt den Einzahlungsbetrag sowie das Einzahlungsdatum. In der Methode einzahlung() erfolgt die Gutschrift der Einzahlung sowie der errechneten Zinsen. Mittels kd1.zeigeGuthabenZins() wird das Ergebnis angezeigt. `Elsa 1510.45 14.965753424657535`

Der Aufruf einer Methode erfolgt nach dem Schema:

Objektvariable.Methode()

beziehungsweise

Objektvariable.Methode(aktuelle Parameter).

Die Definition einer Methode erfolgt nach dem Schema

def nameMethode(self)

beziehungsweise

def nameMethode(self, formale Parameter)

Die Anzahl der formalen Parameter ist wegen des Parameters *self* stets um eins größer als die Anzahl der aktuellen Parameter.

Übungen

In dem obigen Programm Bankkonto sind Verbesserungen vorzunehmen.

- Verbessern Sie das Ausgabeformat von self.guthaben und self.zins.

- Eine wichtige Forderung aus dem Kontovertrag ist nicht erfüllt. Ein Sparkonto wird nur auf Guthabenbasis geführt. Es muss stets ein Mindestguthaben von 1,00 Euro aufweisen. Somit muss bei einer Abhebung geprüft werden, ob eine Auszahlung in der angegebenen Höhe zulässig ist. Nehmen Sie die erforderliche Ergänzung in der Methode *abhebung()* vor!

7.2.4 Indirekter Aufruf eines Objekts

Stellt man die Objekte in eine Liste ein, so kann man ein Objekt indirekt adressieren. In dem nachstehenden Beispiel sind Namen der Objekte in einer Liste *ktoNr* eintragen.

```
kd1 = Sparkonto(458,"Elsa" ,510.45, 12.50)
kd2 = Sparkonto(459, "Fritz", 680.33, 13.85)
kd3 = Sparkonto(467, "Alissa", 1025.23, 89.41)
ktoNr = [kd1, kd2, kd3]
nr = int(input("Sparkonto 0,1 oder 2: "))
ktoNr[nr].zeigeGuthabenZins()
euro = float(input("Euro:"))
ktoNr[nr].einzahlung(euro, date.today())
ktoNr[nr].zeigeGuthabenZins()
```

Wird bei der Eingabeanweisung eine 0 eingetippt, verweist ktoNr[nr] auf den Eintrag kd1. Somit erfolgt der Zugriff auf das Objekt, das mittels der Objektvariablen kd1 adressiert wird. Die folgende Einzahlung und Anzeige betrifft ebenfalls das Objekt kd1.

7.2.5 Kommunikation zwischen Instanzen einer Klasse

Problem

Elsa hat zwei Sparkonten.

```
kd1 = Sparkonto(458,"Elsa" ,510.45, 12.50)
kd3    Sparkonto(857, “Elsa“, 1875.50, 85.40)
```

Sie möchte nun einen bestimmten Betrag vom Konto 857 auf das Konto 458 umbuchen lassen.

Bei dem derzeitigen Entwicklungsstand des Programms müssten erst eine Auszahlung und dann eine Einzahlung vorgenommen werden. Gewünscht

ist, dass die Umbuchung mit einer Anweisung aus dem Hauptprogramm durchgeführt werden kann.

Lösungsidee

Der erste Schritt bei einer Umbuchung besteht in einer Abbuchung. Anschließend muss der Betrag dem anderen Konto gutgeschrieben werden. Dazu muss die Adresse des anderen Kontos bekannt sein, um dort die Einzahlungsmethode aufrufen zu können.

Der Kopf der Methode umbuchen() könnte wie folgt formuliert werden:
def umbuchen(self, adresse, betrag, datum):

Der Aufruf der Methode lautet dann:

```
kd3.umbuchen(kd1, 75.5, date(2021, 10, 15))
```

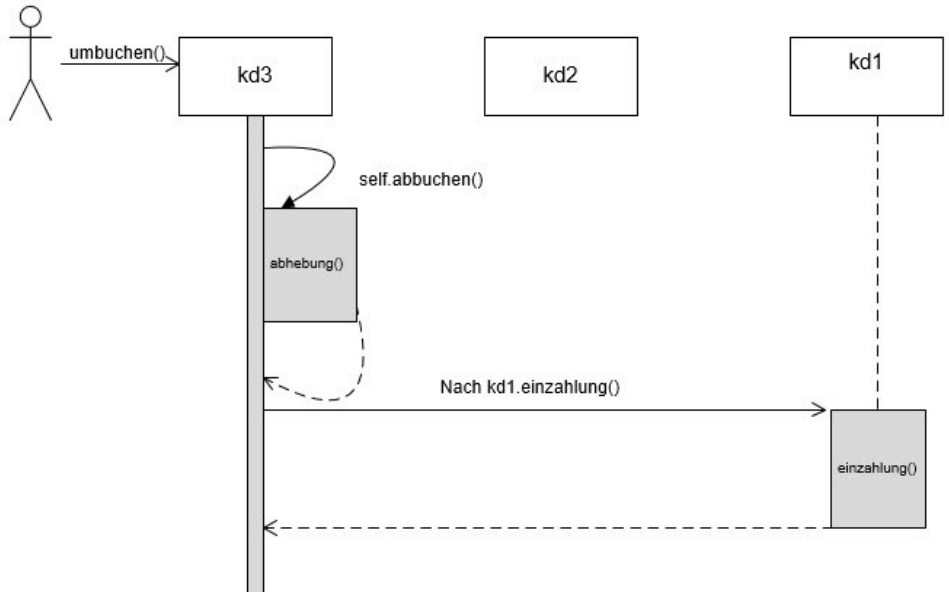
Lösung

Auf diese Weise erhält das Objekt kd3 die Adresse vom Objekt kd1. Die Methode wird dann wie folgt formuliert:

```
def umbuchen(self, adresse, betrag, datum):
    self.abhebung(betrag, datum)
    adresse.einzahlung(betrag, datum)
    adresse.zeigeGuthabenZins()
    self.zeigeGuthabenZins()
```

Die Kommunikation zwischen Objekten erfolgt mittels Botschaften, die in Methoden programmiert werden. Im obigen Beispiel wird beim Aufruf der Methode *kd3.umbuchen(kd1, 75.5, date(2021, 10, 15)* die Adresse der Objektvariablen kd1 an den Parameter *adresse* übergeben. Der Parameter *betrag* nimmt den Umbuchungsbetrag auf, und das Datum wird an den Parameter *datum* übergeben. Der Methodenaufruf *self.abhebung(betrag, datum)* bezieht sich auf *kd3*. Die Methode *adresse.einzahlung(betrag, datum)* adressiert das Objekt *kd1*, da an *adresse* die Adresse von *kd1* übergeben worden ist.

Sequenzdiagramm



7.2.6 Sichtbarkeit: public, private, protected

In dem Summierungsbeispiel wird ein Problem sichtbar. In dem Additionsbefehl (*) wird eine Variable, die ein Attribut repräsentiert, aus dem Hauptprogramm, also von außerhalb des eigenen Objekts angesprochen. Der Zugriff auf die Attribute eines Objektes sollte nur durch objekteigene Methoden gestattet sein. Das ist einer der Grundregeln der objektorientierten Programmierung, das auch als Kapselung bezeichnet wird.

Warum die Kapselung sinnvoll ist, ist bei dem derzeitigen Lernstand noch nicht ersichtlich, weil im Augenblick nur auf einer Klasse gearbeitete wird. Sind in einem Programm mehrere Klassen eingebunden, kann es zu Überschneidungen wegen der Gleichheit von Variablennamen kommen. Außerdem sind bei der Anwendungssoftware Sicherheitsaspekte zu berücksichtigen. Nicht jeder Nutzer darf auf alle Funktionen eines Programms nutzen. Durch eine entsprechende Ausgestaltung der Methode, die den Zugriff auf eine Variable regelt, kann der Zugriff einer nicht berechtigten Person abgewehrt werden.

In Python haben die Variablen im Normalfall die Zugriffseigenschaft public, also öffentlich. Somit kann aus allen Methoden und aus dem Hauptprogramm auf jede Variable zugegriffen werden. Oder anders ausgedrückt: Jede

Variable kann von allen Anweisungen eines Programms „gesehen“ werden. In der Fachsprache spricht man auch von Sichtbarkeit.

Beispiel

```
# zugriffsregelung
from datetime import *
class Sparkonto():

    def __init__(self, kontonr, guthaben):
        self.kontonr = kontonr
        self.guthaben = guthaben

    def zeigeGuthaben(self):
        print(self.kontonr, format(self.guthaben, '.2f'))

    def abhebung(self, auszahlung):
     ...

# Hauptprogramm - main -
kd1 = Sparkonto(458,510.45)
kd2 = Sparkonto(459, 680.33)
kd3 = Sparkonto(467, 1025.23)
kd1.zeigeGuthaben()
print(kd1.guthaben)
kd1.guthaben -=100
print(kd1.guthaben)
```

Werden die drei letzten Anweisungen des Hauptprogramms ausgeführt, wird folgendes Ergebnis angezeigt:

```
458 510.45
510.45
410.45
>>>
```

Mit print(kd1.guthaben) erfolgt ein direkter Zugriff auf das Konto. Um Geld abzubuchen, wird auch keine Methode abhebung() benötigt. Mit kd1.guthaben -= 100 werden direkt aus dem Hauptprogramm 100 Euro abgebucht, also auch ohne Prüfung, ob die Abhebung zulässig ist.

Es ist also gefährlich, die Zugriffseigenschaft der Variablen guthaben auf public zu belassen.

Möchte man den Zugriff auf eine Variable von außerhalb des Objekts blockieren, muss der Zugriff auf *private* gesetzt werden. Durch einen doppelten

Unterstrich vor ihrem Namen erhält eine Variable die Zugriffseigenschaft privat: **self.__guthaben**

```
    def __init__(self, kontonr, guthaben):
        self.kontonr = kontonr
        self.__guthaben = guthaben
```

Erfolgt nun ein Zugriff auf self.__guthaben aus dem Hauptprogramm, meldet das System:

```
Traceback (most recent call last):
  File "c:\users\gk\mu_code\zugriffsregelung.py", line 32, in <module>
    print(kd1.__guthaben)
AttributeError: 'Sparkonto' object has no attribute '__guthaben'
```

Nun ist das Attribut geschützt. Es kann nur aus einer Methode aufgerufen werden, die zum Objekt gehört.

Innerhalb der Methode abhebung() kann auch eine Sicherung gegen unbefugtes Abbuchen eingebaut werden. So kann zum Beispiel die Abbuchung von einem Zugangswort oder Passwort abhängig gemacht werden.

```
    def abhebung(self, auszahlung, geheim):
        if geheim == "Chef":
            if auszahlung <= self.__guthaben:
                self.__guthaben -= auszahlung
                self.zins -= self.zinsenBerechnen(auszahlung, datum)
            else:
                print("Du kannst nur  "+ str(self.guthaben) + " abheben")
        else:
            print("unberechtigter Zugriff")
```

Aufruf aus dem Hauptprogramm:

```
zugangswort = input("Wie lautet das Zugangswort?")
kd1.abhebung(100, zugangswort)
```

Wird ein falsches Zugangswort eingegeben, wird keine Abbuchung vorgenommen. Vielmehr meldet das System

```
458 510.45
Wie lautet das Zugangswort?Vater
unberechtigter Zugriff
```

Zugriffseigenschaften von Methoden

Auch Methoden haben im Allgemeinen die Zugriffseigenschaft public. Um sie auf private umstellen, werden vor den Methodennamen zwei Unterstriche eingefügt.
Beispiel:

def __abhebung(self, auszahlung)

Die Methode dann nicht mehr aus dem Hauptprogramm aufgerufen werden. Vielmehr bedarf es einer weiteren Methode, über die der Zugriff gesteuert wird.

Neben Zugriffseigenschaft privat und public gibt es noch die Eigenschaft protected. Die wird durch einen einfachen Unterstrich vor dem Variablennamen verliehen: self.guthaben. Bei dieser Kennzeichnung bleibt dennoch die Zugriffseigenschaft public erhalten.

7.2.7 Überladen von Methoden

Beispiel

Das Kreditinstitut wirbt damit, dass es zur Einzahlung auf ein Sparkonto auch noch DM-Beträge annimmt.

Ein DM-Betrag kann aber nicht einfach einem Euro-Sparkonto gutgeschrieben werden, da eine DM nur 1/1.5583 Euro entspricht. Die Methode

def einzahlung (self, eingezahlterBetrag, datum)

leistet die Umrechnung nicht. Es müsste eine weitere Methode formuliert werden, in der die Umrechnung erfolgt. Das würde dann zu zwei Einzahlungsmethoden führen. Das ist aber nicht erwünscht, weil diese Änderung noch weitere Änderungen nach sich ziehen würde.

Python kennt den Datentyp None = Nichts. Damit kann das Problem gelöst werden.

```
def einzahlung(self, eingezahlterBetrag, datum, dm=None):
    if dm is not None:
        eingezahlterBetrag /= 1.95583
    self.guthaben += eingezahlterBetrag
    self.zins += self.zinsenBerechnen(eingezahlterBetrag, datum)
```

Methodenaufruf bei einer DM-Einzahlung:

```
kd1.einzahlung(1955.83,date(2022,2,3),1)
```

Bei dieser Gestaltung des Methodenkopfs kann die bisherige Form des Methoden-Aufrufs beibehalten werden. Es ändert sich lediglich der Aufruf bei einer DM-Einzahlung. Dieser enthält dann einen dritten Parameter hinter der Datumsangabe, hier beispielweise eine 1.

In dem Methodenkopf wird dann festgehalten, dass dm nicht mehr None ist. Somit kann in der if-Anweisung abgefragt werden, ob dm nicht None ist. Der Operator lautet **is not**.

Entsprechend dem Ergebnis der Abfrage auf not None erfolgt die Umrechnung auf den Euro-Wert. Die restlichen Anweisungen der Methode bleiben somit unberührt.

7.2.8 Zusammenfassung

- Eine Klasse gib an, über welche Eigenschaften und Methoden die zu generierenden Objekte haben sollen.
- Ein Objekt ist eine Instanz seiner Klasse.
- Definition einer Klasse: class nameDerKlasse:
- Mittels def __init__(self, parameter, ..): wird der Konstruktor definiert.
- Die Objektattribute haben die Form self.varname.
- Eine Methode wird mittels def eingeleitet.
- Der Aufruf einer Klasse erfolgt über ihre Objektvariable.
- Die Kommunikation zwischen zwei Objekten erfolgt mittels Botschaften. Sie werden von einer Methode an eine Methode des anderen Objekts versandt.
- In dem Methodenaufruf der sendenden Methode ist Adresse des Empfängerobjekts anzugeben.
- Objekte können auch indirekt aus einer Liste aufgerufen werden.
- Die Variablen wie auch Methoden erhalten durch zwei vorgesetzte Unterstriche die Eigenschaft privat. Damit wird ein Zugriff von außerhalb des zugehörigen Objekts unterbunden.
- Der None-Typ ermöglicht ein Überladen einer Methode.

7.2.9 Aufgaben

Aufgabe 1

Die Begriffe Klasse, Methoden, Objekt, Instanzen sind zentrale Begriffe in der objektorientierten Programmierung.

a) Klasse und Objekt – in welcher Beziehung stehen sie zueinander?
b) Eine Klasse besteht aus Attributen und Methoden. Wozu dienen Methoden?
c) Welche Aufgabe hat def__init__(self)?
d) Was versteht man unter Kapselung?

Aufgabe 2

In dem Programm 0805_731 fehlt die Methode, mit der das Zinsguthaben kapitalisiert wird. Entwickeln Sie eine geeignete Methode und formulieren Sie den Methodenaufruf.

Ferner soll das Gesamtguthaben durch Summierung der Sparguthaben ermittelt werden.

Aufgabe 3

Das nachstehende Programm Bruttolohn weist eine Reihe von Fehlern auf. Nehmen Sie die erforderlichen Korrekturen vor.

```
class Bruttolohn:

    def init__(self, persnr, hname, stdlohn):

        self.nr = persnr
        self.name = hname
        self.stlo = stdlohn
        self.uebStdLohn = 0

    def berechneBrutto(self, stunden):
        brutto = self.stlo * stunden
        if stunden > 40:
            self.berechneZuschlag(stunden - 40)
        print(self.name, format(brutto, '.2f'), format(self.uebStdLohn, ".2f"))

    def Berechnezuschlag(self, mehrStunden):
        uebStdLohn = mehrStunden * 0,5
        return

pers1 = Bruttolohn(2, "Adam", 20.90)
pers2 = Bruttolohn(3, "Berta", 21.50)
pers1.berechneBrutto(45)
pers2.berechneBrutto(30)
```

Aufgabe 4

Die Firma Klender versorgt Autowerkstätten und Industriebetriebe mit Schmierstoffen.

Entwickeln Sie eine Klasse, mit der das Lager für die Fassware verwaltet werden kann.

Schmierstoffart	Artikelnummer	Maximalbestand	Istbestand	Meldebestand	Preis pro Fass
Hydrauliköl	10058	45	25	8	2560,00 €
Hydrauliköl	50263	40	37	10	1300,00 €
Getriebeöl	36251	60	40	5	985,00 €
Motorenöl	25143	40	8	10	1085,00 €

Es werden Methoden benötigt, mit denen man

e) feststellen kann, ob bei einem Artikel der Meldebestand erreicht ist
f) ermitteln kann, wie viele Fässer nachgekauft werden können unter Berücksichtigung von Maximalbestand und Istbestand.
g) den Verkaufspreis berechnen kann unter Berücksichtigung des Mengenrabatts. Ab einer Bestellmenge von 3 mehr Fässern wird ein Rabatt von 5 % gewährt.
h) die Auslieferung von Fässern buchhalterisch erfassen kann
i) die Einlagerung von neuen Fässern buchen kann.

Aufgabe 5

Der Mühlenbetrieb hat seine Siloanlage von 3 auf 4 Silos vergrößert. Das neue Silo fasst 78000 Doppelzentner. Zwei weitere Anlagen sind in der Planung.

Für die Verwaltung der Silos wurde bislang das nachstehende Programm Siloverwaltung eingesetzt. Ein IT-Beratungsunternehmen empfiehlt, die Siloverwaltung mit Hilfe eines objektorientierten Programms vorzunehmen.

j) Analysieren Sie das Siloverwaltungsprogramm
k) Entwickeln Sie ein OOP-Programm, das dieselbe Funktionalität wie das vorliegende Programm besitzt.
l) Was muss in den Programmen geändert werden, wenn die Anzahl der Silos 10 beträgt.
m) Gefällt Ihnen der OOP- oder der prozedurale Ansatz besser?

Bisheriges Programm:

```
silo = [1, 70000, 35000, 2, 75000, 31000, 3, 68000, 5000]

def siloanzeigen(nr):
    for i in range(0, len(silo), 3):
        if silo[i]== nr:
            print(silo[i+2])

def freieKapazitaet(nr):
    for i in range(0, len(silo), 3):
        if silo[i]== nr:
            freiKap = silo[i+1]- silo[i+2]
            print(freiKap)

def zubuchen(nr, menge):
    for i in range(0, len(silo), 3):
        if silo[i]== nr:
            silo[i+2] += menge
            siloanzeigen(nr)

def abbuchen(nr, menge):
    for i in range(0, len(silo), 3):
        if silo[i]== nr:
            silo[i+2] -= menge
```

```
        siloanzeigen(nr)

s = int(input("Siloinhalt von Silo Nr. "))
siloanzeigen(s)
s = int(input("Freie Kapazität von Silo Nr. "))
freieKapazitaet(s)
s = int(input("Zubuchen zu Silo Nr. "))
m = int(input("Wie viele Doppelzentner?"))
zubuchen(s, m)
s = int(input("Abbuchen zu Silo Nr. "))
m = int(input("Wie viele Doppelzentner?"))
abbuchen(s, m)
```

Aufgabe 6

a) Das OOP-Programm (s. Aufgabe 5) ist zu erweitern:
b) Auf die Bestandsdaten dürfen nur objekteigene Methoden zugreifen
c) Beim Zubuchen muss die Kapazitätsgrenze berücksichtig werden.
d) Bei Abbuchung muss sichergestellt sein, dass mindestens 100 dz im Silo verbleiben.
e) Es muss möglich sein, Mengen von einem Silo in ein anderes umzubuchen.
f) Die Geschäftsleitung möchte wissen, wie hoch der Lagerbestand ist. Die Summierung der Füllstände darf nur von jemandem vorgenommen werden, der über das Passwort „Sicher“ verfügt.

7.3 Vererbung

7.3.1 Eine Klasse einschachteln

Problem

Es liegen zwei Entwürfe für ein Lohnprogramm vor. Das Programm mit der Klasse Angestellte ist ausgelegt für Berechnung des Monatslohns von Angestellten, die ein festes Monatsgehalt erhalten. Das Lohnprogramm für die Klasse Arbeiter hat zu berücksichtigen, dass Arbeiter nach der Anzahl der geleisteten Arbeitsstunden entlohnt werden. Der Monatslohn ergibt sich durch die Multiplikation von Stunden und Lohnsatz.

Lohnprogramm Angestellte

Angestellte
Name Adresse Monatsgehalt
berechne Nettolohn

Lohnprogramm Arbeiter

Arbeiter
Name Adresse Stundenlohnsatz
berechne Monatslohn berechne Nettolohn

Beim Vergleich der beiden Programme stellen Sie sicherlich fest, dass sie hinsichtlich ihrer Struktur fast gleich sind. Der Unterschied besteht darin, dass bei einem Arbeiter der Monatslohn erst noch ermittelt werden muss, bevor der Nettolohn berechnet werden kann.

Der Gedanke liegt dann nahe, dass die beiden Programme bzw. Klassen zu einem Programm bzw. Klasse Mitarbeiter zusammengefasst werden können. Die Klasse *Arbeiter()* bedient sich der Methoden der Klasse *Angestellte*(). Die Klasse Angestellte() ist dann eine Art Oberklasse, die Klasse *Arbeiter*() eine Art Unterklasse.

Die Klasse Angestellte() kann dann unverändert übernommen werden. Die Klasse *Arbeiter()* wird mit den erforderlichen Methoden „eingebaut“.

Programm

```
# 0805_731_ nettolohn
class Angestellte():
    steuersatz = 20.0                                   # 1: Klassenvariable
    sozversi = 18.0

    def __init__(self, name, adresse):
        self.name = name
        self.adresse = adresse
```

```
    def nettolohn (self, monatslohn):
        steuer = self.prozentrechnung(monatslohn, Angestellte.steuersatz)
        sozver = self.prozentrechnung(monatslohn, Angestellte.sozversi)
        netto = monatslohn - steuer - sozver
        print(self.name,self.adresse, format(netto,'.2f') + " Euro")

    def prozentrechnung(self, gw, ps):
        return 0.01*gw*ps

class Arbeiter(Angestellte):                                    # 2: Vererbung
    def __init__(self, name, adresse, lohnsatz):
        super().__init__(name, adresse)                         # 3: super()
        self.lohnsatz = lohnsatz

    def nettolohn(self,stunden):
        Angestellte.nettolohn(self, stunden * self.lohnsatz)


An0 = Angestellte("Natz", "Bochum", )
Ar1 = Arbeiter("Willi", "Essen",20.0)
An0.nettolohn(4000)
Ar1.nettolohn(180)
```

Erläuterungen

 # 1: Klassenvariable

Die beiden Variablen *steuersatz* und *sozvers* bezeichnet man als Klassenvariablen. Sie können aus allen Objekten der Klasse aufgerufen werden durch Voranstellung des Klassennamens.

 # 2: Vererbung

In der Kopfzeile der Klasse Arbeiter wird als Parameter der Name der Klasse angegeben, deren Methode die Klasse *Arbeiter* nutzen möchte. Hier ist die Klasse *Angestellte*.

 # 3: super()

Die Klasse Arbeiter wird durch def __init__(self, name, adresse, lohnsatz): initialisiert. Da diese Anweisung den Konstruktor der Klasse *Angestellte* überschreibt, bleibt er durch super()__init__(name, adresse) erhalten. Zu beachten ist, dass nur die originären Attribute von Angestellte als Parameter aufgeführt werden, also kein self:

```
super().__init__(name, adresse)
```

Ablauf des Programms

Den Ablauf des Programms kann man sich wie folgt vorstellen. Mittels der Anweisung Ar1 = Arbeiter(“Willi“, “Essen“, 20.0) wird eine Instanz der Klasse *Arbeiter* mit diesen Daten eingerichtet.

Das UML-Diagramm zeigt den Zustand des Objekts:

Arbeiter	
Name:	Willi
Adresse:	Essen
Stundenlohnsatz:	20.0
berechne Monatslohn	
berechne Nettolohn	

Die Anweisung *Ar1.nettolohn(180)* ruft die Methode *nettolohn()* in der „Willi“-Instanz auf. Dabei wird die Instanzadresse von “Willi“ sowie die Stundenanzahl übergeben.

Die Instanzadresse kann man sich anzeigen lassen:

```
<__main__.Arbeiter object at 0x000002049211E10>
```

In der Methode *nettolohn*() in der “Willi“-Instanz wird nun die nettolohn-Methode der Klasse *Angestellte* aufgerufen. An sie wird die Adresse der “Willi“-Instanz sowie das Produkt aus Stunden und Lohnsatz übergeben. *lohnsatz* ist ein Attribut der *Arbeiter*-Klasse. Deshalb muss sie mittels *self* als solches qualifiziert werden.

Die Parameter der nettolohn-Methode in der Angestellte-Instanz übernehmen die Adresse der “Willi“-Instanz sowie das Produkt aus Stunden und Lohnsatz als Monatslohn. Nach den Berechnungen werden der Name und der Nettolohn ausgegeben.

```
print(self.name,self.adresse, format(netto,'.2f') + " Euro")
```

Das vorangestellte *self* bei Name und Adresse adressiert die “Willi“-Instanz. Somit wird ausgegeben:

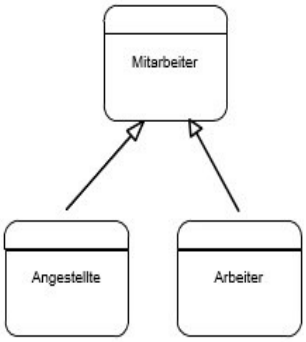
```
Willi Essen 2376.00 Euro
>>>
```

Kritik

Aufgrund der Vererbung braucht man in der Tat nur ein Programm, um das Arbeitsentgelt für Angestellte und Arbeiter zu berechnen. In der vorliegenden Form ist es jedoch nicht akzeptabel, dass bei jeder Monatsabrechnung

für die Angestellten das Monatsgehalt eingeben werden muss. Das Monatsgehalt ist wie auch der Lohnsatz durch den Tarifvertrag oder durch den Arbeitsvertrag für längere Zeit festgelegt. Diese Angaben sind quasi Stammdaten.

Die vorliegende Klassenkonstruktion stellt die Klasse *Arbeiter* als einen Spezialfall der Klasse *Angestellte* dar. Klassen müssen aber in einer Weise konstruiert sein, dass Erweiterungen oder Änderungen leicht eingebaut werden können. In der Klasse *Angestellte* könnten später Sonderzahlungen oder Prämien zu berücksichtigen sein. Bei der Lohnberechnung in der Klasse *Arbeiter* könnten Akkordzulagen wichtig werden. Deshalb ist es sinnvoll, zwei eigenständige Klassen Angestellte und Arbeiter zu bilden. Innerhalb der Klasse wird der Bruttolohn ermittelt. Auf diesen Betrag beziehen sich die Berechnungen für die Steuer und Sozialversicherung.



Die Klasse *Mitarbeiter* stellt die Methoden zur Verfügung, die sowohl von der Klasse *Angestellte* wie auch von der Klasse *Arbeiter* genutzt werden.

Die Attribute der Klasse *Mitarbeiter* sind die Attribute, die beiden Klassen gemeinsam sind, also *name* und *adresse*:

Dann lautet die Klassendefinition für die Klasse *Mitarbeiter*:

```
class Mitarbeiter():
        def __init__(self, name, adresse):
        ...
```

Die Klassendefinition für *Angestellte*:

```
class Angestellte(Mitarbeiter):
        def __init__(self, name, adresse, monatslohn):
        ...
```

Die Klasse *Arbeiter* ist wie folgt zu definieren:

```
class Arbeiter(Mitarbeiter):
        def __init__(name, adresse, lohnsatz)
        ...
```

Durch den Parameter (Mitarbeiter) erben die beiden Klassen die Methoden der Klasse *Mitarbeiter*.

Programm

```
# 0805_734 nettolohnVererbung

class Mitarbeiter():
    steuersatz = 20.0
    sozialvers = 18.0

    def __init__(self, name, adresse):
        self.name = name
        self.adresse = adresse

    def prozentrechnung(self, gw, ps):
        return round(0.01 * gw * ps,2)

    def berechneNettolohn(self):
        brutto = self.monatslohn
        steuer = Mitarbeiter.prozentrechnung(self, brutto, Mitarbeiter.steuersatz)
        sozver = Mitarbeiter.prozentrechnung(self, brutto, Mitarbeiter.sozialvers)
        netto = brutto - steuer - sozver
        print(self.name, self.adresse,
                    '\n\t', "Bruttolohn:   ",format(brutto,'.2f'),
                    '\n\t',  "Lohnsteuer:    ", format(steuer, '.2f'),
                    '\n\t', "Soz-versich.:  ", format(sozver,'.2f'),
                    '\n\t', "Nettolohn:    ", format(netto,'.2f'))

class Arbeiter(Mitarbeiter):
    def __init__(self, name, adresse, lohnsatz):
        super().__init__(name, adresse)
        self.lohnsatz = lohnsatz

    def arbeitslohn(self, stunden):
        self.monatslohn = stunden * self.lohnsatz
        self.berechneNettolohn()

class Angestellter(Mitarbeiter):
    def __init__(self, name, adresse, monatslohn):
        super().__init__(name, adresse)
        self.monatslohn = monatslohn

# ------ main --- Hauptprogramm ----
```

```
Ar1 = Arbeiter("Willi", "Essen", 20.0)
An1 = Angestellter("Cana", "Bochum", 400.0)
An1.berechneNettolohn()
Ar1.arbeitslohn(180)
```

Erläuterungen

Bei der Formulierung der Klassenköpfe ist zu beachten, dass die Initialisierung der Attribute nicht vollständig innerhalb der Klassendefinition erfolgt. Bei der Klasse Angestellte zum Beispiel wird nur das Attribut monatslohn als Objektvariable eingerichtet.

```
    self.monatslohn = monatslohn
```

Die Attribute name und adresse werden per super() in der übergeordnete Klasse Mitarbeiter als Objektvariablen angelegt.

```
def __init__(self, name, adresse):
    self.name = name
    self.adresse = adresse
    # print (name, adresse)
```

Sie können das überprüfen, indem Sie darunter die Anweisung print(name, adresse) setzen. Dann werden bei der Instanziierung der Name und die Adresse angezeigt.

7.3.2 Bildung einer Oberklasse durch Generalisierung

Problem

Die Firma Klender handelt mit Fassöl. Bei der Rechnungserstellung werden zwei Kundenarten unterschieden: Großkunden und Werkstätten. Bei den Großkunden werden Rabatte gewährt, die pro Kunde unterschiedlich sein können. Bei den Werkstätten wird der Listenpreis berechnet zuzüglich Lieferkosten, deren Höhe abhängt vom Lieferbezirk und der Einkaufsmenge.

Anlieferungskosten

Lieferbezirk	Lieferkosten
1	10,00 €
2	20,00 €
3	30,00 €
>3	50,00 €

Werden mehr als 2 Fässer geliefert, entfallen die Anlieferungskosten.

Bei den Großkunden wird pro Fass ein Pfand von 30,00 € berechnet.

Nach der Ermittlung der Verkaufspreise ist ein Verkaufsprotokoll auszugeben:

```
-----Verkaufsprotokoll---------------
Keller 3516.45
Summen 1184.05
Fitten 2169.87
Lippmann 5278.15
```

Eine Problemstellung, deren Lösung einige Überlegungen erfordert. In einem ersten Schritt werden die Attribute zusammengestellt und geprüft, welche übereinstimmen.

Vergleich der Attribute

Großkunden	Werkstätten
Kundennummer Name Adresse Rabattsatz	Kundennummer Name Adresse Lieferbezirk

Generalisieren

Die Klasse Großkunden und die Klasse Werkstätten weisen Gemeinsamkeiten, aber auch Unterschiedlichkeiten auf. Unter Generalisieren versteht man die Zusammenfassung der der Gemeinsamkeiten in einer Art Oberklasse. Die Unterschiedlichkeiten werden in Unterklassen eingebracht.

Wie aus den obigen Datensätzen zu erkennen ist, treten die Attribute Kundennummer, Name und Adresse in beiden Klassen auf. Diese drei Attribute können der Oberklasse zugeordnet werden.

Methoden

In beiden Klassen sind Berechnungen vorzunehmen.

Warenwert = Menge * Einzelpreis

Bei beiden Kundenarten sind Prozentwerte zu berechnen.

Bei der Berechnung des Rabattbetrages
Bei der Berechnung der Umsatzsteuer

Die Berechnung des Verkaufspreises ist in den beiden Klassen unterschiedlich

Bei Werkstätten:

Verkaufspreis = Warenwert +Lieferkosten + Umsatzsteuer
Werden mehr als 2 Fässer gekauft, werden keine Lieferkosten berechnet.
Lieferkosten werden in einer komplexen if-Anweisung ermittelt. Deshalb wird dafür eine eigene Methode gewählt.

Bei Großkunden:

Verkaufspreis = Warenwert – Rabatt + Pfand + Umsatzsteuer
Rabatt = Warenwert* Rabattsatz/100
Pfand = Menge * Pfandbetrag
Umsatzsteuer = (Listenpreis * Menge + Rabatt + Pfand) *Umsatzsteuersatz/100

Schließlich soll ein Verkaufsprotokoll erstellt werden. Weil das für beide Klassen gilt, sind die dazu erforderlichen Anweisungen in der Oberklasse zu formulieren. Der zurzeit gültige Umsatzsteuersatz und der Fasspreis werden als Klassenvariablen definiert. Ebenso die Liste, die das Verkaufsprotokoll aufnimmt.

Fasst man die gemeinsamen Attribute und Methoden zusammen, ergibt sich folgende Oberklasse Kunde:

- Name
- Adresse
- Warenwertberechnung
- Berechnungen durchführen: Umsatzsteuer, Verkaufspreis
- Anzeigen Kundendaten, Rechnungsdaten

Die Klasse Großkunde umfasst zusätzlich folgende Attribute und Methoden:

- Rabattsatz
- Pfand als Konstante

- Preisberechnen
- (Bruttopreis, Rabatt, Pfand, Umsatzsteuer, Verkaufspreis)

Die Klasse Werkstatt verfügt über die zusätzlichen Attribute und Methoden:

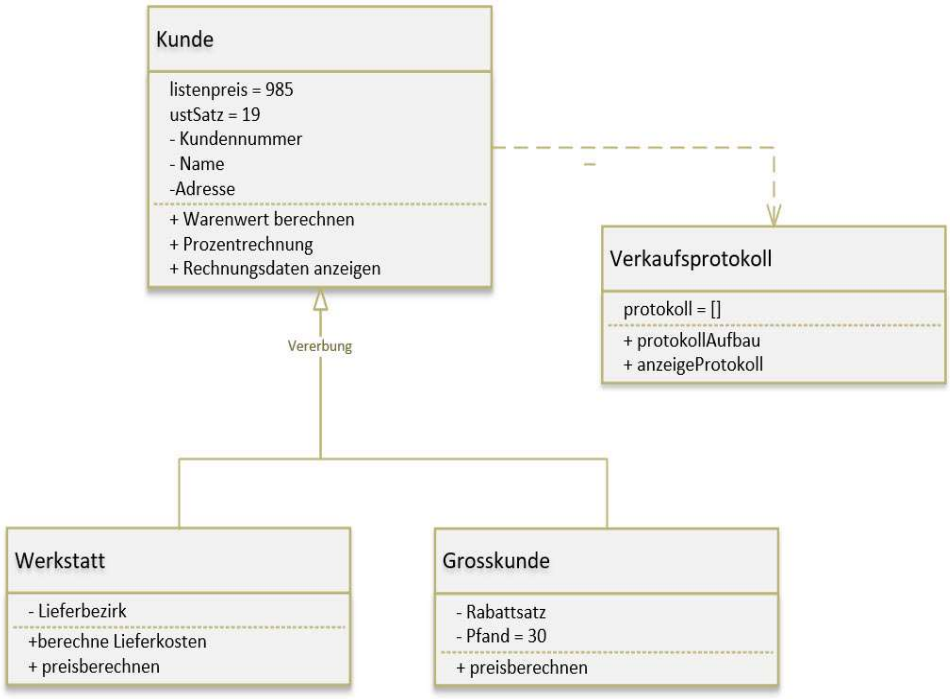
- Lieferbezirk
- Berechnung der Lieferkosten
- Preisberechnen
- (Bruttopreis, Lieferkosten, Umsatzsteuer, Verkaufspreis)

Verkaufsprotokoll

Nach Abschluss der Ermittlung der Verkaufspreise ist das Verkaufsprotokoll auszugeben. Diese Aufgabe wird in einer gesonderten Klasse (vkProtokoll) programmiert. Diese ist nicht in die Vererbung eingebunden. Ihre Aufgabe ist es, den Kundennamen und den Verkaufspreis in eine Liste einzutragen und nach Abschluss der Verkaufspreisberechnungen den Inhalt der Liste auszugeben.

UML

Die Darstellung in einem UML-Klassendiagramm verdeutlicht das Prinzip von Oberklasse und Unterklasse.



Grafisch wird die Unter-Oberklassenbeziehung durch einen transparenten Dreieckspfeil kenntlich gemacht. Er soll ausdrücken, dass die Attribute und Methoden der Oberklasse den Unterklassen zur Verfügung stehen. Es gibt keinen Hinweis von der Oberklasse zu den Unterklassen. Die Oberklasse weiß nicht, dass sie eine Oberklasse ist. Sie weiß auch nicht, wie viele Unterklassen zu ihr gehören.

Programm

```
#0805_ 732 Vererbung

class Kunde():
    einzelPreis = 985.0                                    # Klassenvariable
    ustSatz = 19

    def __init__(self, kundennr, name, adresse):
        self.knr = kundennr
        self.nam = name
        self.adr = adresse

    def prozentrechnung(gw, ps):
        return round(0.01 * gw * ps,2)

    def warenwertBerechnen(self, men, preis):
        return round(men * preis,2)

    def anzeigen(self, np, k, u, vkp):
        print(self.knr, self.nam, self.adr)
        print("%7.2f" % np, '\n', "%6.2f" % k, '\n',
              "%6.2f" % u, '\n'"%7.2f" % vkp)
        vkProt.protokollAufbau(self.nam, vkp)

class werkstatt(Kunde):
    def __init__(self, kundennr, name, adresse, bezirk):
        super().__init__(kundennr, name, adresse)          #
        self.bez = bezirk

    def lieferkosten(self):
        if self.bez == 1:
            liefkosten = 10
        elif self.bez == 2:
            liefkosten = 20
        elif self.bez == 3:
```

```
            liefkosten = 35
        else:
            liefKosten = 50
        return liefkosten

    def preisberechnen(self, menge):
        if menge > 2:
            anlieferung = 0
        else:
            anlieferung = self.lieferkosten()
        nettopreis = Kunde.warenwertBerechnen(self, Kunde.einzelPreis, menge)
                        + anlieferung
        umsatzsteuer = Kunde.prozentrechnung(nettopreis, Kunde.ustSatz)
        vkPreis = nettopreis + umsatzsteuer
        Kunde.anzeigen(self, nettopreis, anlieferung, umsatzsteuer, vkPreis)

class grossKd(Kunde):
    def __init__(self, kundennr, name, adresse, rabattsatz):
        super().__init__(kundennr, name, adresse)
        self.rabsatz = rabattsatz
        self.pfand = 30.0

    def preisberechnen(self, menge):
        bruttopreis = Kunde.warenwertBerechnen(self, Kunde.einzelPreis,
                          menge)
        rabatt = Kunde.prozentrechnung(bruttopreis, self.rabsatz)
        nettoPreis = bruttopreis - rabatt
        umsatzsteuer = Kunde.prozentrechnung(nettoPreis, Kunde.ustSatz)
        pfand = self.pfand * menge
        vkPreis = nettoPreis + umsatzsteuer + pfand
        Kunde.anzeigen(self, bruttopreis, rabatt, pfand, vkPreis)

class vkProtokoll:
    protokoll = []

    def anzeigeProtokoll(self):
        print("-----Verkaufsprotokoll----------")
        end = len(self.protokoll)
        for x in range(0, end, 2):
            print(self.protokoll[x], self.protokoll[x+1])

    def protokollAufbau(self, name, betrag):
        self.protokoll.append(name)
        self.protokoll.append(betrag)
```

```
# ---main-----
vkProt = vkProtokoll()
w1 = werkstatt("w1125", "Keller", "Essen", 2)
w2 = werkstatt("w2587", "Summen","Aalen", 1)
f1 = grossKd("f1965", "Fitten", "Frankfurt", 10)
f2 = grossKd("f2632", "Lippmann", "Ochtrup", 12.5)
w1.preisberechnen(3)
w2.preisberechnen(1)
f1.preisberechnen(2)
f2.preisberechnen(5)
vkProt.anzeigeProtokoll()
```

Klassenvereinbarungen

```
class Kunde():
   einzelPreis = 985.0
   ustSatz = 19
```

Der Einzelpreis und der Umsatzsteuersatz sind für alle Kundenarten gleich. Durch die Positionierung der Variablen *einzelPreis* und ustSatz direkt nach der Klassenüberschrift sind sie für alle Teile des Programms gültig. Man charakterisiert sie deshalb auch als Klassenvariablen im Gegensatz zu den Instanzvariablen, die nur innerhalb einer Instanz gültig sind.

Die Formulierung der Klasse *Kunde* deutet nicht darauf hin, dass sie eine Oberklasse ist. Das ergibt sich erst aus der Formulierung der Unterklassen.

```
class werkstattKunde):
   def __init__(self,kundennr, name, adresse, bezirk):
      super().__init__(kundennr, name, adresse)
```

Durch den Parameter *werkstatt(Kunde*) erbt die Klasse *werkstatt* die Methoden der Klasse *Kunde*. Mit def__init__(self, …) wird auch deren Konstruktor überschrieben. Durch super()__init__ (…) wird das verhindert.

Bei der Klasse *grossKd* liegt dieselbe Konstruktion vor:

```
class grossKdKunde):
   def __init__(self,kundennr, name, adresse, rabattsatz, skontosatz):
      super().__init__(kundennr, name, adresse)
```

Eine Klasse wird durch zwei Angaben zu einer Unterklasse.

1. Angabe der Oberklasse als Parameter im Klassenkopf

   class name(nameOberklasse)

2. Super-Konstruktor mit Angabe der Attribute der Oberklasse, aber ohne self.

Kommunikation zwischen den Klassen

In der Klasse *grossKd* erfolgt die Berechnung des Bruttopreises mit Hilfe der Methode *warenwertBerechnen*(). Diese Methode ist in der Oberklasse definiert. Damit das System die Methode findet, muss die Klassenbezeichnung vorangesetzt werden. Das gilt auch für die Klassenkonstante einzelPreis.

```
bruttopreis = Kunde.warenwertBerechnen( self, Kunde.einzelPreis , menge)
```

Der Parameter *self* muss angegeben werden, damit das System weiß, wer die Methode aufruft und das Ergebnis erhält. Die Variable *menge* gilt nur innerhalb der Methode *preisberechnen*().

In der nächsten Anweisung werden in der Parameterliste zwei Attribute an Methode *Kunde.prozentrechnung()* übergeben.

```
rabatt = Kunde.prozentrechnung(bruttopreis, self.rabsatz)
```

Die Variable *bruttopreis* wird innerhalb der Methode preisberechnen() gebildet und braucht nicht weiter qualifiziert zu werden. Bei *rabsatz* muss *self*. vorangesetzt werden, weil *rabsatz* der Name eines Attributes ist.

Zur Anzeige der Verkaufsdaten werden die berechneten Werte an die Methode *anzeigen()* der Oberklasse übergeben.

```
Kunde.anzeigen(self, bruttopreis, rabatt, pfand, vkPreis)
```

In der Klasse *werkstatt* muss zunächst geprüft werden, ob Kosten für die Anlieferung in Rechnung gestellt werden müssen.

```
def preisberechnen(self, menge):
    if menge > 2:
        anlieferung = 0
    else:
        anlieferung = self.lieferkosten()
```

Falls die bestellte Menge nicht größer 2 ist, wird die Methode *lieferkosten*() aufgerufen, in der die Kostenbetrag für die Anlieferung in Abhängigkeit vom Verkaufsbezirk ermittelt wird.

Die weiteren Berechnungen erfolgen nach demselben Prinzip wie in der Klasse *grossKd*.

Formatierung

Die Anzeige der Rechnungsdaten übernimmt die Oberklasse. Hier wurde eine weitere Möglichkeit der Formatierung von Ausgaben verwendet.
"%5.2f"%umsatzsteuer

Die Formatierungvorschrift steht in Anführungs- und Prozentzeichen: “% x,yf“%. Das x steht für die Anzahl der Zeichen, aus denen die Zahl besteht, inklusive Dezimalpunkt. Das y gibt die Anzahl der Nachkommastellen an. Das f steht für den Zahlentyp float.

Für die Ausgabe einer Ganzzahl lautet die Formatierung „“%xd“%. Das x steht die Anzahl der Ziffern, aus der die Zahl besteht, d kennzeichnet die Zahl als Ganzzahl. Die Variable, deren Wert auszugeben ist, steht direkt hinter der Formatangabe.

Verkaufsprotokoll

Das Objekt Verkaufsprotokoll erhält seine Daten von der Methode anzeigen() in der Klasse Kunde

```
vkProt.protokollAufbau(self.nam, vkp)          {Klasse Kunde}

def protokollAufbau(self, name, betrag):       {Klasse vkProtokoll}
```

Beim Anfügen mittels append ist zu berücksichtigen, dass append immer nur ein Argument zulässt.

Die Anzeige des Listeninhalts erfolgt in einer for-Anweisung.

```
laenge = len(self.protokoll)
for x in range(0, laenge, 2):
    print(self.protokoll[x], self.protokoll[x+1])
```

Die Länge der Liste wird mit der len-Funktion festgestellt.

Die Laufvariable x beginnt bei 0. Die Schrittweite beträgt 2, weil der Protokolleintrag aus den beiden Daten Kundenname und Verkaufspreis besteht.

main - Hauptprogramm

Das Objekt *vkProtokoll* muss vor dem Aufruf der Preisberechnungen gebildet werden, damit die Klasse *Kunde* die Adresse kennt, zu der sie die Protokolldaten senden soll.

Mit der letzten Anweisung im Hauptprogramm wird die Ausgabe der Protokolldaten aufgerufen.

7.3.3 Zusammenfassung

- Bei Vererbung kann die Unterklasse auf die Attribute und Methoden der Oberklasse zugreifen.
- Klassenvariablen stehen sowohl der Ober- wie auch der Unterklasse zur Verfügung.

- Instanzvariablen sind nur innerhalb der Instanz gültig.
- Eine Klasse wird zur Unterklasse, indem bei der Klassenvereinbarung der Namen der Oberklasse als Parameter angegeben wird. Class unter-Klasse(oberKlasse):
- In super():__init__(parameter) werden nur die originären Attribute der Oberklasse genannt.
- Attribute der Oberklasse werden in den Unterklassen nicht angelegt.
- Man spricht von Generalisierung, wenn gemeinsame Attribute der Unterklassen in der Oberklasse eingestellt werden.
- Beim Aufruf von Methoden, die in einer anderen Klasse definiert sind, muss der entsprechende Klassennamen vorangestellt werden.
- In dem Methodenaufruf muss auch self angegeben werden, damit das System weiß, woher der Aufruf stammt.
- Eine formatierte Ausgabe von Zahlen erreicht man durch
- “%anzahlziffern,anzahlnachkommaf“%.
  Das f steht für float. Bei Ganzzahlen steht anstelle des f ein d.

7.3.4 Aufgaben

Aufgabe 1

Die Firma Elsa verkauft Motorenöle in zwei Formen:

Fassware: Dass Fass fasst 50 Liter. Pro Fass wird eine Pfandgebühr von 30,00 Euro erhoben. Auf den Ölpreis wird ein Nachlass von 5 % gewährt.

Gebinde: Ein Gebinde besteht aus 6 Dosen à 1.5 Liter.

Der Ölpreis beträgt 5.25 Euro

Der Nettopreis ist auszugeben nach Eingabe der Fassanzahl oder Gebindeanzahl.

Aufgabe 2

Das Lohnprogramm ist zu überarbeiten: Bei den Angestellten ist zu berücksichtigen, dass ihnen eine Prämie gezahlt werden kann. Bei den Arbeitern muss die Überstundenvergütung eingebaut werden. Überstunden liegen vor, wenn mehr als 175 Stunden gearbeitet wurde. Der Überstundenzuschlag beträgt 15 %.

Aufgabe 3

In der Firma Klender werden die Firmenfahrzeuge auch privat genutzt. Deshalb werden Fahrtenbücher geführt, die auch für die Kostenabrechnung benötigt werden. Bei Dienstfahrten wird festgehalten:

- Fahrer, gefahrene Kilometer.
- Bei Privatfahrten wird notiert:
- Fahrer, gefahrene Kilometer, verbrauchter Kraftstoff.

Bei Privatnutzung muss der Fahrer 0,30 Euro pro gefahrenen Kilometer sowie 1,45 Euro pro Liter Kraftstoff bezahlen.

Bei Dienstfahrten wird ein kalkulatorischer Kostenfaktor von 2,50 Euro in Ansatz gebracht.

a) Entwickeln Sie ein UML-Klassendiagramm

b) Schreiben Sie ein Programm, mit dem die Fahrtkosten berechnet werden können.

8 Graphische Bedienoberfläche

8.1 Zielsetzung

Ein Programm muss rasch und möglichst fehlerfrei genutzt werden können. Heute gibt es kaum noch Programme in der Praxis, mit denen nicht unter Einsatz einer graphischen Bedienoberfläche gearbeitet wird. Die Bedienoberfläche, kurz GUI für Graphical User Interface, kann die Programmbedienung vereinfachen, wenn sie benutzerorientiert aufgebaut ist.

Die Bedienoberfläche setzt sich aus mehreren graphischen Elementen zusammen, die Widgets genannt werden. Python stellt mit Tkinter eine Vielzahl von Widgets zur Verfügung: Rahmen, Ein- und Ausgabefelder, Schaltflächen, Radio- und Checkbuttons.

Bei Einsatz von GUI ändert sich die Art der Kommunikation zwischen Mensch und Computer. Bei einer graphischen Bedienoberfläche bestimmt der Benutzer, wann ein Programmlauf beginnt. In der GUI ist mindestens eine Schaltfläche (Button) einzubauen, mittels der der Benutzer die Verarbeitung der Daten startet. Das geschieht in der Regel durch ein Click-Ereignis. Im Programm muss beschrieben sein, was passieren soll, wenn eine Schaltfläche geklickt wurde. In der Fachsprache spricht man von einem event-driven-program bzw. von einer ereignisorientierten Programmierung.

Ein ereignisorientiertes Programm besteht aus mindestens 2 Teilen. In einem Teil wird der Aufbau des Bildschirmformulars beschrieben. In einem weiteren Teil sind die Anweisungen formuliert, die aufgrund bestimmter Ereignisse auszuführen sind.

Beispiel

Bei der Verwaltung von Sparkonten dient die Bildschirmmaske der Eingabe von Ein- bzw. Auszahlungsbeträgen. Die Verarbeitung der Eingabedaten erfolgt nach Klicken einer Schaltfläche, über die der Benutzer dem Computer anzeigt, dass die Eingabe abgeschlossen ist.

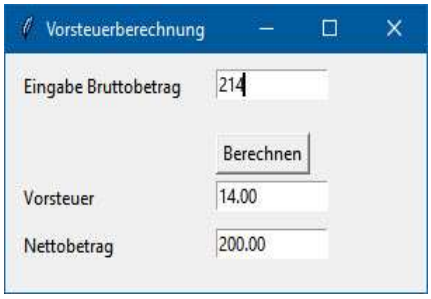
Für Python gibt es eine Reihe von Generatoren, mit denen Bedienoberflächen „bequem“ erstellt werden können. Für den Einstieg in die GUI-Programmierung ist Tkinter gut geeignet, weil es sich auf das Wesentliche beschränkt.

8.2 Aufbau einer Bildschirmmaske

8.2.1 Programmbeispiel

Wie eine Bedienoberfläche aufgebaut wird, soll an einem kleinen Beispiel dargelegt werden. In ihm wird die Umsatzsteuer aus dem Rechnungsbetrag errechnet. Normalerweise wird auf den Rechnungen die Umsatzsteuer direkt ausgewiesen. Bei Fahrkarten zum Beispiel ist das nicht immer der Fall. Somit muss bei Fahrtkostenabrechnungen in der Buchhaltung die Umsatzsteuer aus dem Ticketpreis ermittelt werden. Der Umsatzsteuersatz beträgt 7 %.

Bedienoberfläche




```
from tkinter import *
#0805_821.pyw
class Vorsteuerberechnung:
    ustSatz = 0.07
    def __init__(self):

        self.fenster = Tk()                                                    #1
        self.brutto = DoubleVar()                                              #2
        self.ust = DoubleVar()
        self.netto = DoubleVar()
        self.fenster.title('Vorsteuerberechnung')                              #3
        self.fenster.geometry('300x150+100+150')                               #4

        Label(self.fenster,text='Eingabe Bruttobetrag').place(x=10,y=10)       #5
        Entry(self.fenster,textvariable = self.brutto).place(x=150,y=10,width=80)  #6
        Button(self.fenster, text='Berechnen',
                                          =self.rechnung).place(x=150,y=50)    #7
        Label(self.fenster,text='Vorsteuer').place(x=10,y=80)
        Entry(self.fenster,textvariable=self.ust).place(x=150,y=80,width=8     #8
        Label(self.fenster, text='Nettobetrag').place(x=10, y=110)
        Entry(textvariable = self.netto).place(x=150, y=110,width=80)          #9
```

```
        self.fenster.mainloop()                                          #10

    def rechnung(self):
        r_brutto=self.brutto.get()                                       #11
        r_netto= r_brutto/(1+Vorsteuerberechnung.ustSatz)
        r_steuer = format(r_brutto-r_netto, '.2f')
        self.ust.set(r_steuer)                                           #12
        self.netto.set(value=format(r_netto,'.2f'))

u = Vorsteuerberechnung()
```

Erläuterungen

Mittels from tkinter import * wird das Grafikmodul eingeladen.

#1 fenster = Tk(): Die Klasse Tk() wird instanziiert. Der Objektname ist hier *fenster*.

#2 In den folgenden drei Zeilen werden die Steuerelemente für die Ein- und Ausgaben vereinbart. Man nennt sie **Steuerelementvariablen**. Ihr Datentyp ist entweder IntVar(), DoubleVar() oder StringVar(). Der Zugriff auf diese Steuerelemente erfolgt durch set() und get(), wie weiter unten dargelegt wird.

#3 self.fenster.title('bezeichnung') legt den Titel des Fensters fest.

#4 Die Größe des Fensters wird durch eine Zeichenkette der Form geometry('Breite x Höhe') bestimmt. Möchten Sie die Position auf dem Bildschirm festlegen, geben Sie die Koordinaten in folgender Weise an: ('Breite x Höhe +100+20'). Das Fenster wird, ausgehend von der oberen linken Bildschirmecke um 100 Pixel nach rechts und 20 Pixel nach unten platziert

#5 Mit Label wird ein Text ausgegeben.
Label(self.fenster.text='beliebiger konstanter Text').place(x,y).
Die place-Methode setzt den Text in das Fenster an die angegebene Stelle.

#6 Mit Entry wird ein Ein-bzw. Ausgabefeld festgelegt. Dem Parameter textvariable wird der Name der Steuerelementvariablen zugewiesen. Die place-Methode sorgt für die Platzierung in dem Fenster. Mit width=Anzahl Pixel geben Sie die Länge des Ein- bzw. Ausgabefeldes an.

#7 Eine Schaltfläche wird mit Button eingerichtet. Die Beschriftung der Schaltfläche wird hinter text = 'nameSchaltfläche' angegeben. Hinter command wird die Methode angegeben, die beim Klick auf die

Schaltfläche ausgeführt werden soll. Hier ist es die Methode *rechnung*().

#8 Das zweite Ein-Ausgabefeld wird der Steuerelementvariablen *self.ust* zugeordnert.

#9 In das Ein-Ausgabefeld wird bei der Programmausführung der Inhalt der Steuerelementvariablen *self.netto* ausgegeben.

#10 self.fenster.mainloop() aktiviert das Fenster und bringt es auf den Bildschirm.

#11 In der Methode *rechnung()* werden die erforderlichen Berechnung vorgenommen. Mittels der get()-Methode wird der Inhalt von *self.brutto* ausgelesen und steht nun in *r_brutto* zu Verfügung.

#12 Das Ergebnis der Steuerberechnung wird mittels set() in das zweite Entry-Feld ausgegeben.

#13 Die Methode set() setzt den Wert von *r_netto* in Steuerelementvariable *self.netto*. In Verbindung der Label #9 erscheint der Wert im Fenster.

8.2.2 Widgets

In dem Fenster *Vorsteuerberechnung* wurden drei Widgets verwendet. Dieser Begriff setzt sich aus den Worten Windows und Gadgets zusammen. Gadget kann man mit Gerät übersetzen.

Textzeile:

```
Label(
        self.fenster
        Bedienerführung: text= "Bedienerführung"
        Größe: width = x; height y
        Schrift: font = ('Schriftart',Schriftgröße)
        Vordergrund: fg='Farbe')
```

Als Farbe können die englischen Bezeichnungen wie blue, green usw. eingesetzt werden.

Auch der Hintergrund kann mittels bg = 'Farbe' farblich gestaltet werden.

Eingabe bzw. Ausgabezeile:

```
Entry(
        self.fenster,
        textvariable = self.var)
```

Die Steuerelementvariable self.var muss im Definitionsteil entweder als IntVar(), DoubleVar() oder als StringVar() deklariert worden sein.

Soll der Wert, der in einem Entry-Feld angezeigt wird, im Programm weiter verarbeitet werden, muss er aus der Anzeige in eine Variable übernommen werden. Das bezeichnet man als Auslesen. Dazu wird die Methode get() eingesetzt. Das Schreiben in ein Entry-Feld erfolgt mittels der Methode set().

Schaltfläche:

```
Button(
        self.fenster,
        text = 'Beschriftung',
        command = self.Funktionsaufruf)
```

Die Schaltfläche wird mit der linken Maustaste geklickt. Dieses Ereignis ruft die angegebene Funktion auf.

place(x,y):

Die Methode place() ist der Layoutmanager. Sie sorgt dafür, dass das Widget an der angegebenen Stelle in das Fenster platziert wird. Python kennt noch weitere Layoutmanager wie pack() und grid(), auf die hier aber nicht eingegangen wird.

Ein paar Zwischenfragen:

- Welchen Datentyp können die Ein- bzw. Ausgabefelder in einer Bildschirmmaske haben?
- Den Ein- bzw. Ausgabefeldern werden Variablennamen zugeordnet. Wie bezeichnet man sie?
- Wozu dienen ein Entry- und ein Label-Widget?
- Wie wird in der geometry die Größe und Lage der Bildschirmmaske festgelegt?
- Was bedeutet varA =fenster.zahl.get()
- Wie lautet die Anweisung, um den Inhalt der Variabeln varA in das Entry-Feld zahlB auszugeben?
- Wie und wo werden die Ein- und Ausgabefelder einer Bildschirmmaske vereinbart?
- Wie und wo werden die Ein- und Ausgabefelder einer Bildschirmmaske vereinbart?

8.2.3 Ereignisorientierung

In dem obigen Programm wird die Berechnung von Vorsteuer und Nettobetrag erst durch das Klicken der Schaltfläche *Berechnen* in Gang gesetzt. Hinter dem Schlüsselwort command ist der Name der Funktion anzugeben, in der die Berechnungen durchgeführt werden.

Das Klicken einer Schaltfläche ist wohl das am häufigsten benutzte Ereignis in der Programmierung. Die Anzahl der programmierbaren Ereignisse ist aber größer. Hier einige Beispiele

Aktivität auf dem Bildschirm	Event
Ein Widget hat den Focus erhalten bzw. verloren	FocusIn, FocusOut
Eine bestimmte Taste wurde gedrückt bzw.losgelassen	KeyPress, KeyRelease
Der Mauszeiger ist auf ein Widget gestellt bzw. hat den Bereich des Widgets verlassen.	Enter, Leave
In einer Combobox wurde ein Eintrag ausgewählt.	ComboboxSelected

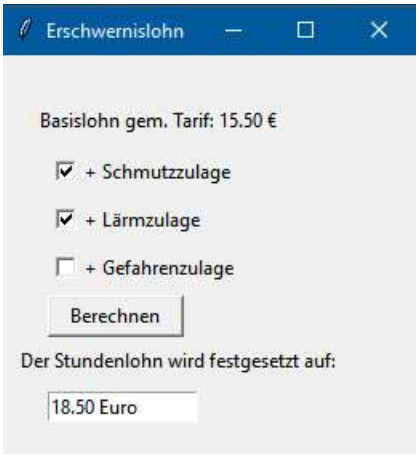
8.3 Auswahl-Schaltflächen

8.3.1 Checkbutton

Benutzerfreundlichkeit ist ein wichtiges Ziel bei der Entwicklung der grafischen Oberfläche. Das Eintippen von Daten mittels der Tastatur kann verringert werden durch den Einsatz von Auswahl-Elementen, die durch Anklicken aktiviert werden.

Beispiel

Bei der Festsetzung des Stundenlohns sind neben dem tariflichen Basislohn die Zulagen zu berücksichtigen. Eine Lärmzulage wird bezahlt, wenn der Arbeitsplatz starkem Lärm ausgesetzt ist. Ist die Arbeit mit viel Schmutz verbunden, wird eine Schmutzzulage vergütet. Eine Gefahrenzulage wird gewährt, wenn die Arbeitsverrichtung mit erhöhten Gefahren verbunden ist, wie z. B. bei der Feuerwehr.



Ausgehend vom tariflichen Basislohn ist der Stundenlohn unter Berücksichtigung von Erschwernissen zu bestimmen. Es können mehrere Zulagen gewährt werden.

Programm

```
from tkinter import *
#0805_831
class Erschwernislohn:
    basislohn=15.5
    def __init__(self):
        self.fenster = Tk()
        self.schmutz=DoubleVar()
        self.laerm=DoubleVar()
        self.gefahr = DoubleVar()
        self.stdLohn = DoubleVar()
        self.fenster.title("Erschwernislohn")
        self.fenster.geometry('260x250+100+150')

        Label(self.fenster,text= '    Basislohn gem. Tarif: 15.50 €     ').
                                    place(x=10, y=30)
        Checkbutton(self.fenster, text='+ Schmutzzulage   ',
                                   variable=self.schmutz).place(x=30, y=60)
        Checkbutton(self.fenster, text='+ Lärmzulage',
                                    variable = self.laerm).place(x=30, y = 90)
        Checkbutton(self.fenster, text='+ Gefahrenzulage  ',
                                     variable=self.gefahr).place(x=30, y=120)
        Button(self.fenster, text='   Berechnen   ', command=
```

```
                                        self.rechnung).place(x=30,y=150)
        Label(self.fenster, text='Der Stundenlohn wird festgesetzt auf:').
                                                    place(x=10, y = 180)
        Entry(self.fenster, textvariable=self.stdLohn, width=15).
                                                      place(x=30,y=210)

        self.fenster.mainloop()

    def rechnung(self):
        zulage = 0
        if self.schmutz.get()==1:
            zulage += 1.5
        if self.laerm.get() ==1:
            zulage+= 1.5
        if self.gefahr.get()==1:
            zulage += 2.5
        self.stdLohn.set(value=format(Erschwernislohn.basislohn
                                                    + zulage,'.2f')+' Euro')

s=Erschwernislohn()
```

Erläuterungen

Das Schlüsselwort zur Vereinbarung eines Auswahlknopfes ist Checkbutton. Mit *self.fenster* wird angegeben, zu welchem Bildschirm er gehört. Darauf folgt die Angabe des Textes, mit dem seine Bedeutung erklärt wird. Schließlich wird die ihm Kontrollvariable zugewiesen, die im Definitionsteil als Steuerelementvariable vereinbart sein muss.

```
        Checkbutton(self.fenster, text='+ Schmutzzulage   ',
                                        variable=self.schmutz).place(x=30, y=60)
```

Der Zustand eines Checkbutton ist entweder 0 oder 1, *onvalue* oder offvalue. Er wird in seiner Kontrollvariablen gespeichert, zum Beispiel *self.schmutz*.

Der Zustand der Kontrollvariablen wird hier in der Funktion *rechnung(self)* überprüft, die durch einen Klick auf die Schaltfläche *Rechnen* aufgerufen wird. In einer if-Konstruktion wird festgestellt, welcher Checkbutton aktiviert wurde. Ist der Wert gleich 1, wird der Wert in *zulage* um den entsprechenden Betrag erhöht.

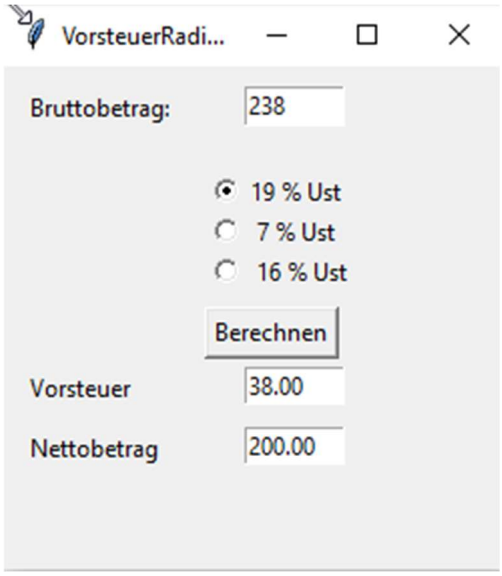
Das Widget Checkbutton ist anzuwenden bei Auswahlantworten, von denen eine oder mehrere Angaben ausgewählt werden können.

8.3.2 Radiobutton

Checkbutton erlaubt die Mehrfachauswahl. Bei Radiobutton werden zwar auch Auswahlantworten angeboten. Es kann aber nur eine Antwort ausgewählt werden.

Beispiel

Aus einem Bruttobetrag sind die Vorsteuer sowie der Nettobetrag zu ermitteln. Der Steuersatz ist entweder 19 %, 7 % oder 16 % (Corona-Zeit). Von den drei Möglichkeiten kann immer nur eine ausgewählt werden.



Bei einer solchen Problemstellung ist das Radiobutton das geeignete Widget. Es lässt immer nur eine Auswahl zu.

Programm

```
from tkinter import *
#0805_832
class radiobt_Vorsteuer:

    def __init__(self):

        self.fenster = Tk()
        self.brutto = StringVar()
        self.ust = StringVar()
        self.netto = StringVar()
        self.auswahl = StringVar()                                    #1
```

```
        self.auswahl.set(19)                                                    #1
        self.fenster.title('Vorsteuer')
        self.fenster.geometry('250x250+450+350')

        lab1=Label(self.fenster,text='Bruttobetrag:')
        lab1.place(x=10, y=10)
        e1=Entry(self.fenster,textvariable=self.brutto)
        e1.place(x=120, y=10, width=50,height=20)
        rb1=Radiobutton(self.fenster,text='19 % Ust', variable=
                                        self.auswahl, value=19)                 #2
        rb1.place(x=100, y = 50)
        rb2=Radiobutton(self.fenster, text=' 7 % Ust', variable=self.auswahl,
                                                        value=7)                #2
        rb2.place(x=100, y=70)
        rb3=Radiobutton(self.fenster, text=' 16 % Ust', variable=self.auswahl,
                                                      value=16)                 #2
        rb3.place(x=100, y=90)
        bt=Button(self.fenster, text='Berechnen',command=self.rechnung)  #3
        bt.place(x=100,y=120)
        lab3=Label(self.fenster,text='Vorsteuer')
        lab3.place(x=10,y=150)
        e2=Entry(self.fenster,textvariable=self.ust)
        e2.place(x=120,y=150,width=50)
        lab4=Label(self.fenster, text='Nettobetrag')
        lab4.place(x=10,y=180)
        e3=Entry(self.fenster,textvariable = self.netto)
        e3.place(x=120,y=180,width=50)

        self.fenster.mainloop()

    def rechnung(self):                                                         #3
        r_brutto=float(self.brutto.get())
        r_steuersatz = float(self.auswahl.get())
        r_netto= r_brutto/(1+r_steuersatz*0.01)
        r_steuer = format(r_brutto-r_netto, '.2f')
        self.ust.set(r_steuer)
        self.netto.set(value=format(r_netto,'.2f'))

rb = radiobt_Vorsteuer()
```

Erläuterungen

In diesem Programmbeispiel wurden die Widget-Vereinbarungen und ihre Platzierungen getrennt formuliert.

```
    lab1=Label(self.fenster,text='Bruttobetrag:')
    lab1.place(x=10, y=10)
```

#1 Im Definitionsteil wird die Variable, die das Auswahlergebnis aufnehmen soll, vereinbart und mit einem Wert vorbelegt. Hier wurde mittels der set-Methode der Wert 19 vorgegeben.

```
    self.auswahl.set(19)
```

Die Vorbesetzung ist erforderlich!

#2 Die Vereinbarung der Radiobutton ähnelt der des Checkbutton. Im Unterschied zum Checkbutton wird der Auswahl-Click aber in nur einer einzigen Variablen festgehalten, die dann mit dem zugeordneten Wert belegt wird.

```
    Radiobutton(self.fenster, text=' 5 % Ust', variable=self.auswahl,
                value=7)                                              #2
```

Beim Klick auf den Button mit dem Text 7 % USt wird der Variablen *auswahl* der Wert 7 zugewiesen.

#3 Der Klick auf die Schaltfläche *Berechnen* ruft die Funktion *rechnen* auf, in der die Werte für Vorsteuer und Nettobetrag berechnet und ausgegeben werden.

8.3.3 Combobox

Eine Combobox ist ein Listenfeld, aus dem man einen Eintrag auswählen kann.

Beispiel
Die Firma Klender versorgt bundesweit Großkunden mit Fassöl. Für jeden Transport sind die Transportkosten zu berechnen. Ein Fass Öl wiegt 92 kg bzw. 0,092 t. Der Preis pro 1 Tonnenkilometer beträgt 1,50 Euro.

Zur Bedienweise: Man gibt die Anzahl der Fässer ein und klickt auf den Listenfeldpfeil. Damit öffnet man die Ortsliste und wählt mit einem Klick den Bestimmungsort aus. Bei dem Bestimmungsort ist die Entfernung auch schon eingetragen.

Progamm

```
from tkinter import *
from tkinter import ttk                                               #1
```

```
#0805_833
class Transportkosten:
    def __init__(self):
        self.fs = Tk()
        self.km = DoubleVar()
        self.anzahl = DoubleVar()
        self.transportPreis = DoubleVar()
        self.fs.title("Transportpreis")
        self.fs.geometry('300x150+100+150')

        l1 = Label(self.fs, text='Eingabe Anzahl Fässer').place(x=10, y=10)
        e1 = Entry(self.fs, textvariable=self.anzahl).place(x=150,
                                                            y=10,width=50)
        self.anzahl.set('')
        self.cbo = ttk.Combobox(self.fs,                                    #2
                    values=["Aachen,120","Bielefeld,120", "Essen,30",
                            "Hamm,75", "Kassel,200"])
        self.cbo.place(x=100, y=40, width=100)
        self.cbo.current(0)                                                 #3
        self.cbo.bind("<<ComboboxSelected>>", self.auswahl)                 #4
        l2 = Label(self.fs, text="Kilometer").place(x=10, y=80)
        e2 = Entry(self.fs, textvariable=self.km).place(x=150, y=80, width=50)
        l3 = Label(self.fs, text="Transportpreis").place(x=10, y=100)
        e3 = Entry(self.fs, textvariable=self.transportPreis)
        e3.place(x=150, y=100, width=50)

        self.fs.mainloop()

    def auswahl(self, event):
        ort=self.cbo.get()                                                  #5
        x_ort = ort.split(',')                                              #6
        z_km = x_ort[1]                                                     #7
        self.km.set(z_km)                                                   #8
        gew = self.anzahl.get()*.092                                        #9
        preis = float(z_km) * float(gew)*1.5                               #10
        self.transportPreis.set(value=format(preis,'.2f'))                 #11
```

Erläuterungen

1 Die benötigten Bibliotheken werden eingeladen.

2 Vereinbarung der Combobox: ttk.Combobox. Hinter values wird die Liste der Einträge notiert. [“Eintrag_1“, “Eintrag_2“,…“Eintrag_n“].

Jeder Eintrag ist in Anführungszeichen oder Hochkomma eingeschlossen. Er kann aus mehreren Bestandteilen bestehen, die hier durch Komma getrennt sind.

3 Mittels der Methode current() wird festgelegt, welcher Eintrag neben dem Listenfeldpfeil angezeigt werden soll. Die Combobox wird aber erst „aktiv“, wenn der Listenfeldpfeil geklickt wird.

4 Mit einem Klick auf einen Eintrag in der aufgeklappten Liste wird das gewünschte Listenelement ausgewählt. Dieser Klick löst ein Ereignis aus. Was aufgrund des Ereignisses passieren soll, wird in einer sogenannten Callback-Funktion behandelt. Diese Funktion hat hier den Namen *auswahl(self,event).*

Mit cbo.bind wird das Ereignis <<ComboboxSelected>> an die Methode, hier *auswahl* gebunden.

5 Der ausgewählte Eintrag wird mit cbo.get() hier in eine Variable *ort* ausgelesen.

6 Das Listenelement enthält neben dem Städtenamen auch die Entfernungskilometer. Um an die Kilometerangabe „heranzukommen“, muss der Inhalt des Listenelements aufgesplittet werden.

# 5: ort = self.cbo.get()	# 6: x_ort = ort.split(‘,‘)	# 7: z_km = x_ort[1]
Kassel,200	[‘Kassel‘,‘200‘]	200

8: Die Kilometerangabe wird in das entsprechende Entry-Feld gesetzt, dem self.km zugeordnet ist.

9 Die Anzahl der Fässer wird ausgelesen und mit dem Fassgewicht multipliziert.

#10 Die Tonnenkilometer z_km * gew wird mit dem Tonnenkilometerpreis multipliziert.

#11 Der Transportpreis wird in die Bildschirmmaske gesetzt.

In diesem Programm wird deutlich, dass mittels der Methode bind Ereignisse mit Methoden bzw. Funktionen verknüpft werden.

8.3.4 Messagebox

Für die Kommunikation zwischen dem Nutzer und dem Programm stellt Python Messageboxen zur Verfügung. Eine Messagebox ist ein kleines Fenster, in dem das Programm dem Anwender Hinweise geben kann. Solche Hinweise können zum Beispiel Fehlerhinweise oder eine Aufforderung zur Bestätigung sein.

Die Abbildung zeigt, dass alle Messageboxen eine Kopfzeile sowie einen Hinweistext aufweisen, die im Programm zu formulieren sind.

Messageboxen unterscheiden sich hinsichtlich der Schaltflächen. Bei Einsatz einer Box mit nur einer Schaltfläche kann man das Programm anhalten und durch Klick auf die Schaltfläche *ok* das Programm fortsetzen. Weist die Box zwei oder drei Schaltflächen auf, kann man die Programmsteuerung beeinflussen.

Die obigen Messageboxen wurden mit dem nachstehenden Programm erstellt.

```
#0805_834
from tkinter import messagebox
info=messagebox.showinfo("Information", "Mit OK geht es weiter")        #1
print("Die Information war gut")
rep = messagebox.askyesno("Abfrage","Du hast die Wahl")                 #2
if rep == True:
   print("Ja geklickt")
else:
   print("Nein geklickt")
antwort = messagebox.askyesnocancel('3 Möglichkeiten',
                                    'welcher der 3 soll es sein')       #3
if antwort == True:
   print ('Du hast auf Ja geklickt')
elif antwort == False:
   print('Du hast auf Nein geklickt')
else:
   print ('Du hast auf Abbrechen geklickt')

response = messagebox.showerror("Fehlermeldung", "Hier liegt ein Fehler vor!")
```

```
                                                                    #4
if response == 'ok':
   print('Programmzeile 17')
```

Erläuterungen

#1 Bei showinfo wird das Programm angehalten und nach einem Klick auf die Schaltfläche OK fortgesetzt.

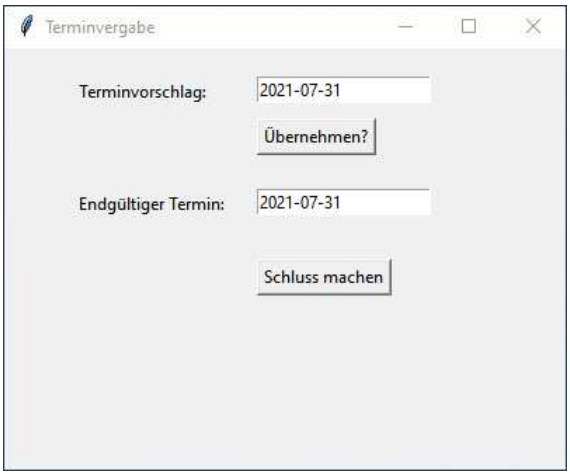
#2 Die Box askyesno hat die Schaltflächen *Ja* und *Nein*, die abgefragt werden können. Die Schaltfläche *Ja* liefert den Wert *True*. Die Schaltfläche *Nein* hat den Rückgabewert *False*. Das Ergebnis der Klickaktion wird in der Variablen *antwort* gespeichert und kann dort abgefragt werden.

#3 Die Box askyesnocancel weist drei Schaltflächen auf. Welche Schaltfläche geklickt wurde, wird mittels einer if-elif-else-Konstruktion ermittelt.

#4 Die beiden Messageboxen showerror wie auch showwarning weisen jeweils nur eine Schaltfläche auf. Sie unterscheiden sich durch das Symbol. Das Klicken der OK-Schaltfläche kann abfragt werden.

8.3.5 Messageboxen in GUI-Programm

Das folgende Programm zeigt, wie Messageboxen in ein Programm mit grafischer Bedienoberfläche eingebaut werden. Das Programm behandelt ein Teilproblem einer Terminvergabe. Das System macht einen Terminvorschlag. Der Nutzer hat die Möglichkeit, den Termin zu übernehmen oder einen anderen Termin einzugeben.



Programm

```
from tkinter import*
from tkinter import messagebox                                      #1
from datetime import date                                           #2
```

```
#0805_835
class Terminvergabe:

    def __init__(self):
        self.ws = Tk()
        self.ws.title('Terminvergabe')
        self.ws.geometry('400x300')
        self.terminVorschlag = StringVar()
        self.terminEndgueltig = StringVar()

        Label(self.ws,text= 'Terminvorschlag:').place(x=50, y=20)          #3
        Label(self.ws, text='Endgültiger Termin:').place(x=50, y=100)      #3
        e1 = Entry(self.ws, textvariable=self.terminVorschlag)             #4
        e1.place(x=180, y=20)
        self.terminVorschlag.set(date.today())                             #5
        e2 = Entry(self.ws, textvariable=self.terminEndgueltig)            #4
        e2.place(x=180, y=100)

        btn1 = Button(self.ws,text='Übernehmen?',command=
                                        self.askQuestion)                  #6
        btn1.place(x=180, y=50)

        btn2 = Button(self.ws, text='Schluss machen',
                                    command=self.askYesNo)                 #6
        btn2.place(x=180, y=150)

        self.ws.mainloop()

    def askQuestion(self):                                                 #7
        reply = messagebox.askyesno('Bestätigung', 'Datum übernehmen')
        if reply == True:
            messagebox.showinfo('Gut', 'Datum wird übernommen!')
            b_datum=self.terminVorschlag.get()
            self.terminEndgueltig.set(b_datum)
        else:
            messagebox.showinfo('Änderung', 'Gib einen neuen Terminvorschlag ein!')
            self.terminVorschlag.set("   ")

    def askYesNo(self):                                                    #8
        reply = messagebox.askyesno('Nachfrage',
                                        'Willst du Programm beenden?')
        if reply == True:
            messagebox.showinfo('Schluss', 'Ende des Programms')
            self.ws.destroy()
```

```
        else:
            messagebox.showinfo('', 'Gut, dass du weitermachst!')

t = Terminvergabe()
```

Erläuterungen

#1 Die erforderliche Bibliothek wird eingeladen.

#2 Für den Aufruf des aktuellen Datums wird datetime benötigt.

#3 Beschriftung für die Entry-Widgets

#4 Vereinbarung der Entry-Widgets

#5 Dem Entry-Widget Terminvorschlag wird mit das aktuelle Datum durch date.today() zugewiesen.

#6 Die beiden Schaltflächen werden vereinbart.

#7 Die Funktion aksQuestion(self) wird aufgerufen, wenn die Schaltfläche *Übernehmen* geklickt wird. In der Variablen *reply* wird festgehalten, welche Schaltfläche geklickt wurde. Bei *Ja* wird das aktuelle Datum in die Variable *b_datum* ausgelesen und in das Entry-Feld *Engültiger Termin* geschrieben. Wurde die *Nein*-Schaltfläche geklickt, wird das Entry-Feld *Terminvorschlag* „leer geschrieben“ mit dem Hinweis, einen neuen Terminvorschlag einzugeben.

#8 Der Klick auf die Schaltfläche *Schluss machen* ruft die Funktion askYesNo(self) auf. Dort wird nachgefragt, ob man das Programm beenden möchte. Wurde die Schaltfläche *Ja* geklickt, wird die Information ausgegeben, dass das Programm beendet wird. Mit self.ws.destroy() wird das Programm beendet und die Bedienoberfläche verschwindet vom Bildschirm.

8.4 Zusammenfassung

- Eine graphische Bedienoberfläche soll die Nutzung eines Programms vereinfachen.
- Die graphischen Elemente werden Widgets genannt.
- Die Bedienoberfläche wird programmtechnisch als Klasse angelegt, die durch die Methode mainloop() dauerhaft angezeigt wird.
- Die Felder für Ein- und Ausgaben nennt man Steuerelementvariablen, die entweder vom Typ IntVar(), DoubleVar() oder StringVar() sind.
- Mittels der Methode geometry wird die Größe des Fensters festgelegt.

- Die Methode place(x,y) plaziert die Texte und Steuerelemente auf die Bedienoberfläche.
- Mittels Entry () werden die Eingabe- und Ausgabefelder gebildet.
- Label ist für Textfelder zuständig.
- Mittels der Button-Methode werden Schaltflächen gebildet.
- Die Methode set() wird benötigt, um einen Wert bzw. Text in ein Entry-Feld zu schreiben.
- Die Methode get() liest den Inhalt eines Feldes aus.
- Bei einem ereignisorientierten Programm zählen zu den wichtigen Ereignisse FocusIn, FocusOut, KeyPress, KeyRelease, Enter, Leave sowie ComboboxSelected.
- Häufig benutzte Schaltflächen sind Button, Checkbutton, Radiobutton, Combobox.
- Zur Kommunikation zwischen Benutzer und Programm werden Messageboxen eingesetzt.
- Die durch die Steuerelemente ausgelösten Ereignisse werden außerhalb der Fenster-Klasse in Methoden behandelt. Man nennt sie Event-handler.

8.5 Aufgaben

Aufgabe 1

Bei Spezialölen werden die Mengenangaben oftmals in oz, also Unzen gemacht. Die Mitarbeiter haben dann das Problem der Umrechnung in Liter. Um ihnen die Arbeit zu erleichtern, hat Alex ein Umrechnungsprogramm geschrieben. Aber irgendwie läuft das Programm nicht. (1 oz = 473,18 Milliliter)

```
#0805_85A1_Umrechnung von oz nach Liter
from tkinter import *

class Wasistfalsch:
    def __init__(self):

        self.fenster = Tk()
        self.milliliter = DoubleVar
        self.unze = DoubleVar
        self.fenster.title("Liter-Unzen-Umrechnung")
        self.fenster.geometry('300x100+100+150')
        # 16 unzen = 473,18 Milliliter
```

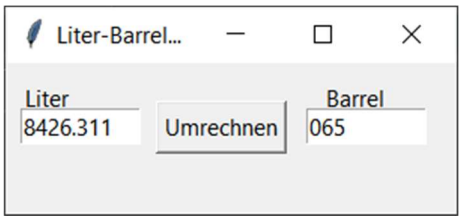
```
        l1 = Label(self.fnster,text="Unzen").place(x=10,y=10)
        e1 = Entry(fenster,textvariable=self.unze).place(x=10,y=30, width=80)
        btn = Button(self.fenster, text="Umrechnen", command = self.rechnen)
        btn.place(10,55)
        l2=Label(self.fenster,text="liter").place(x=70,y=100)
        e2 = Entry(self.fenster, self.millliter).place(x=170,y=130,width=80)

        self.fenster.mainloop()

    def rechnen(self):

        m_oz=self.unzen.get()
        m_milliter=1000*m_oz*473.18/16
        self.milliliter.set(value=format(m_milliter,'.3f'))

p = Wasistfalsch
```

Korrigieren Sie die Fehler und bringen Sie das Programm zum Laufen.

Aufgabe 2

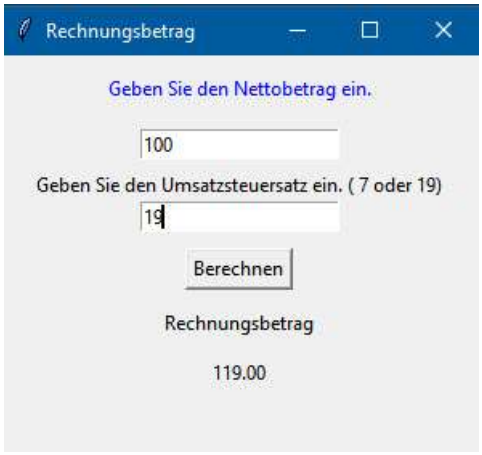
In der Firma besteht häufig das Problem, dass Literangaben in Barrel und umgekehrt Barrel in Literangaben umzurechnen sind.

Arbeitsablauf: Soll eine Literangabe in Barrel umgerechnet werden, gibt man die Anzahl in das Feld Liter ein und klickt auf <Umrechnen>. Das Ergebnis wird im Feld Barrel angezeigt. Entsprechend ist das Verfahren bei der Umrechnung von Barrel auf Liter. (1 Barrel = 158,987 Liter)



Aufgabe 3

Entwickeln Sie aus dem Beispielprogramm Vorsteuerberechnung ein Programm, mit dem nach Eingabe des Nettobetrags und des Umsatzsteuersatzes der Rechnungsbetrag berechnet wird.



Aufgabe 4

In einer Kantine kostet die Tasse Kaffee 1,60 Euro. Zusätzlich kann man zusätzlich Sahne, kleines Gebäck, großes Gebäck und Sahnekuchen bestellen.

Zusatz	Sahnehaube	Kleines Gebäck	Großes Gebäck	Sahnekuchen
Preis in Euro	0,40	1,20	2,00	3,00

Erstellen Sie ein benutzerfreundliches Programm, mit dem das Büffetpersonal den Verkaufspreis bestimmen kann.

Aufgabe 5

Die DHL berechnet das Porto für Pakete nach dem Gewicht.

Gewicht	Bis 2 kg	Bis 5 kg	Bis 10 kg	Bis 31,5 kg
Porto	5,00	6,00	8,50	16,50

Nach dem Umsatzsteuergesetz sind Pakete bis 10 kg von der Umsatzsteuer befreit. Ist das Paket schwerer als 10 kg, muss auf das Porto 19 % Umsatzsteuer entrichtet werden. Entwickeln Sie ein benutzerfreundliches Programm.

Aufgabe 6

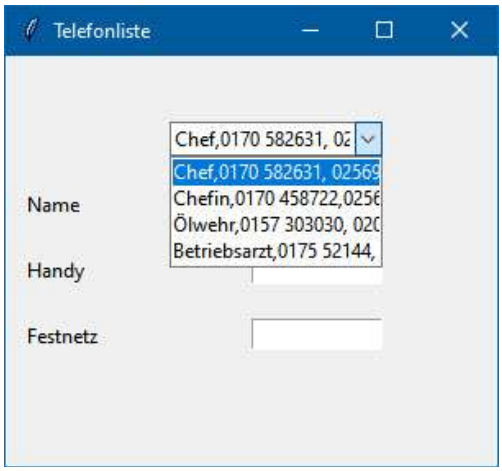
Das Unternehmen zahlt seinen Mitarbeitern freiwillig eine Kinderzulage. Die Zulage ist gestaffelt nach der Anzahl der Kinder.

Anzahl Kinder	1	2	3	4	5 und mehr
Kinderzulage	30,00	70,00	110,00	155,00	200,00

Entwickeln Sie eine Bildschirmmaske, in der nach Eingabe des Bruttolohns und der Anzahl der Kinder der Monatslohn berechnet wird.

Aufgabe 7

Man benötigt im Betrieb eine Telefonliste. Sie soll wie folgt gestaltet sein:



Die Liste besteht aus den Elementen Name, Handy-Nummer, Festnetznummer.

Aufgabe 8

Das Programm Vorsteuer ist zu überarbeiten. Es soll dagegen abgesichert werden, dass aus einem negativen Bruttobetrag die Vorsteuer berechnet wird. Falls der Bruttobetrag negativ ist, soll der Nutzer eine entsprechende Information erhalten und zur erneuten Eingabe aufgefordert werden.

8.6 Exkurs: GUI und Kontoführung

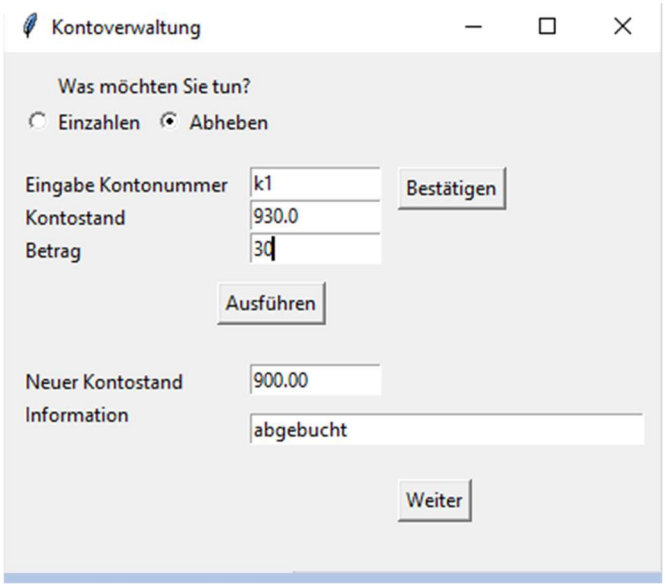
Das nachstehende Programmsystem besteht aus einem GUI-Programm sowie aus einer Klasse Konto, mit der auch ein sogenanntes Kinderkonto verwaltet wird. Das Kinderkonto soll stets ein Mindestguthaben von 100 Euro aufweisen.

Bedienweise

Mittels Radiobutton wird festgelegt, ob es sich bei dem Geschäftsvorfall um Einzahlung oder Abhebung handelt. Die Kontonummer wird eingegeben, hier begrenzt auf die Kontonummern k1, k2 oder ki1. Die Eingabe wird mit der Schaltfläche *Bestätigen* bestätigt. Das löst die Kontosuche aus, als deren Ergebnis der Kontostand oder die Meldung „falsche Kontonummer“ angezeigt wird.

Nach Eingabe des Betrags und dem Klick auf die Schaltfläche *Ausführen* wird der Betrag gebucht und der neue Kontostand angezeigt.

Für weitere Geschäftsvorfälle wird die Schaltfläche *Weiter* geklickt.



Programm

```
# bankabrechnung
from OOP_Konto import Konto, bieneMayaKonto
from tkinter import *
  # 0805_86
class Kontoverwaltung():
    def __init__(self):

        self.fenster = Tk()
        self.fenster.title('Kontoverwaltung')
        self.fenster.geometry('400x300+100+150')
        self.kontonr = StringVar()
        self.kontostand = StringVar()
        self.betrag = StringVar()
        self.vorgang = StringVar()
        self.information = StringVar()
        self.vorgang.set('a')
        self.kontostand_neu = DoubleVar()
        self.gui_kontostand= DoubleVar()
        self.knr = Konto                        # Objekt vereinbaren
        k1 = Konto("k1", 1050)                  # Konten vereinbaren
        k2 = Konto("k2", 274.25)
        ki1=bieneMayaKonto("ki1", 220)
        print("zugelassene Konten: k1, k2, ki1")
        self.kontoliste = [k1, k2,ki1]
        h = 20  # Verschiebemass
        Label(self.fenster, text='Was möchten Sie tun?').place(x=30, y=10)
        rbe=Radiobutton(self.fenster, text="Einzahlen", variable=self.vorgang, value="e").place(x=10, y=30)
        rba=Radiobutton(self.fenster, text="Abheben", variable=self.vorgang, value="a").place(x=90, y=30)
        Label(self.fenster, text='Eingabe Kontonummer').place(x=10, y=50 + h)
        Entry(self.fenster, textvariable=self.kontonr).place(x=150, y=50 + h, width=80)
        Button(self.fenster, text="Bestätigen", command=self.kontoSuchen).place(x=240, y=50 + h)
        Label(self.fenster, text='Kontostand').place(x=10, y=70 + h)
        Entry(self.fenster, textvariable=self.kontostand).place(x=150, y=70 + h, width=80)
        self.kontostand.set("")
        Label(self.fenster, text='Betrag').place(x=10, y=90 + h)
        Entry(textvariable=self.betrag).place(x=150, y=90 + h, width=80)
        self.betrag.set("")
        Button(self.fenster, text='Ausführen', command=self.rechnung).place(x=130, y=120 + h)
        Label(self.fenster, text='Neuer Kontostand').place(x=10, y=170 + h)
        Entry(textvariable=self.kontostand_neu).place(x=150, y=170 + h, width=80)
        Label(self.fenster,text = 'Information').place(x=10,y=190 +h)
        Entry(textvariable=self.information).place(x=150,y=200+h, width=240)
        Button(self.fenster, text="Weiter", command=self.naechster).place(x=240, y=240 + h)


        self.fenster.mainloop()
    def naechster(self):
        self.information.set("")
        self.kontostand_neu.set("")
        self.betrag.set("")
        self.kontostand.set("")
        self.kontonr.set("")

    def rechnung(self):
        # Abhebung,Ausfuehrung nach Kommando Ausfuehren
        if self.vorgang.get() == "a":
            betr = float(self.betrag.get())
            kstand, bemerkung = self.knr.geld_abheben(betr)
            self.kontostand_neu.set(format(kstand,'.2f'))
            self.information.set(bemerkung)

        else:
            betr = float(self.betrag.get())
            kstand , bemerkung = self.knr.geld_einzahlen(betr)
            self.kontostand_neu.set(format(kstand,'.2f'))   # Anzeige in GUI
            self.information.set(bemerkung)
```

```
    def kontoSuchen(self):
        # Aufruf durch Kommando Bestätigen, Einlesen des Kontostandes
        gui_Konto = self.kontonr.get()
        zulaessigeKonten=[]
        for self.knr in self.kontoliste:
            zulaessigeKonten.append(self.knr.kontonummer)

        if gui_Konto in zulaessigeKonten:               # Prüfen, ob Kontonummer zulässig ist
            for self.knr in self.kontoliste:
                if gui_Konto == self.knr.kontonummer:
                    self.kontostand.set(self.knr.kontostand_anzeigen())
                    break
                else:
                    continue
        else:
            self.information.set("Kontonummer falsch")


if __name__ == '__main__':
    kv = Kontoverwaltung()
```

```
# klasse Konto und bieneMayaKonto
class Konto ():

    def __init__(self, kontonummer, kontostand):
        self.kontonummer = kontonummer
        self.kontostand = kontostand
        print(kontonummer, kontostand)

    def geld_abheben(self, betrag):
        self.kontostand -= betrag
        return self.kontostand, "abgebucht"

    def geld_einzahlen(self, betrag):
        self.kontostand += betrag
        return self.kontostand, "gebucht"

    def kontonummer_liefern(self):
        return self.kontonummer

    def kontostand_anzeigen(self):
        #print("aktueller Kontostand: ", self.kontostand)
        return self.kontostand

    def kontostand_aktuell(self):
        return self.kontostand

class bieneMayaKonto(Konto):
    def __init__(self, kontonummer, kontostand):
        super().__init__(kontonummer, kontostand)

    def geld_abheben(self,betrag):
        if self.kontostand - betrag >=100:
            self.kontostand -= betrag
            return self.kontostand, "Gebucht"
        else:
            return self.kontostand,"Mindestguthaben unterschritten"

```

Erläuterungen

Das Programm besteht aus zwei Klassen: *Kontoverwaltung* und *Konto*. Bei der Ausführung von Kontoverwaltung braucht die Klasse *Konto* nicht in den Editor geladen werden, wenn sie dasselbe Verzeichnis wie die *Kontoverwaltung* gespeichert wurde.

Die Steuerung des Programmablaufs erfolgt aus dem Programm *Kontoverwaltung* bzw. der GUI:

In den Zeilen 19 – 23 werden in der Klasse *Kontoverwaltung* drei Instanzen der Klasse Konto angelegt. In den Zeilen 25 – 44 wird der Aufbau des Fensters beschrieben.

Der Klick auf den Button *Bestätigen* ruft das Methode *kontoSuchen ()* auf. In Zeile 72 wird die Kontonummer ausgelesen. In den Zeilen 74 und 75 wird ermittelt, welche Kontonummern zulässig bzw. welche Instanzen der Klasse Konto angelegt worden sind. Existiert das Konto, wird aus der entsprechenden Instanz der Kontostand abgerufen und in der GUI angezeigt.

Das Methode *rechnung*() (Zeilen 79- 68) wird durch Klicken der Schaltfläche *Ausführen* aufgerufen. Der Geldbetrag wird ausgelesen und in Abhängigkeit vor der Vorgangseinstellung zu- oder abgebucht. Dazu wird entweder die Methode *geld_einzahlen()* oder *geld-abheben()* aufgerufen. Als Rückmeldung liefern die Methode den neuen Kontostand sowie eine Bemerkung. Sie werden in den beiden Variablen *kstand* und *bemerkung* gespeichert und anschließend in der GUI angezeigt.

In der Methode *naechster()* wird die GUI „leer“ geschrieben, wenn die Schaltfläche *Weiter* geklickt wurde.

9 Ausnahmen behandeln: try-except

9.1 Fatale Fehler- Programmabsturz

Beispiel

Wenn Python die Anweisung

c = 10 / 2,5

ausführen soll, ist das Ergebnis nicht 4, sondern eine Fehlermeldung

```
    c= a/b
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Es liegt hier ein ValueError vor. Der Datentyp eines Operanden ist, bezogen auf die Rechenanweisung keine Zahl. Der Eintrag 2,5 wird wegen des Dezimalkommas nicht als Zahl erkannt. In der EDV dient der Dezimalpunkt als Dezimaltrennzeichen.

Bei dem Versuch, die Rechenanweisung auszuführen, stürzt das Programm ab. Es handelt sich um einen fatalen Fehler. Auch bei dem Versuch, eine Zahl durch 0 zu dividieren, wird das Programm abstürzen. Es meldet:

```
ZeroDivisionError: division by zero
```

Weitere fatale Ereignisse können bei Dateioperationen auftreten. So könnten zum Beispiel der Name der Datei falsch geschrieben oder ein Dateisatz korrumpiert sein.

9.2 try-exept-Konstruktion

Um Programmabstürze abzufangen, kleidet man die Programmanweisung, bei der es zum Absturz kommen kann, in eine try-except-Konstruktion ein.

```
try:
        Anweisungsblock
except:
        Anweisungsblock
```

Try heißt versuchen. Das System soll versuchen, eine Anweisung auszuführen. Falls das nicht gelingt, soll er die Anweisungen des except-Blocks bearbeiten. Die try-except-Konstruktion funktioniert also wie eine Auswahlanweisung.

Beispiel

```
# 0805_92_1
while True:
   try:
```

```
        a = float(input ("Gib eine Dezimalzahl ein: "))
        b = float(input ("Gib eine weitere Dezimalzahl ein: "))
        c =a/b
        print ("Ergebnis: " + str(c))
        break
    except ValueError:
        print("Dezimalpunkt, kein Dezimalkomma!")
        antwort = input("Fehleingabe bestätigen")
        continue
```

```
Führe aus: try_except1.py
Gib eine Dezimalzahl ein: 5,3
Dezimalpunkt, kein Dezimalkomma!
Fehleingabe bestätigen
```

Aus dem Beispiel ist zu erkennen, dass Python die Eingabe kontrolliert. Die Eingabe 5,3 stellt keine Zahl dar. Somit ist ein Ausnahmefall gegeben. Python springt unmittelbar in den except-Block und führt die dortigen Anweisungen aus.

Das Programm fängt jetzt den Fehler auf und vermeidet einen Absturz. Aufgrund der while-Konstruktion kann die Eingabe wiederholt werden.

Unbefriedigend ist, dass das Programm stehen bleibt, wenn als zweite Zahl eine 0 eingetippt wurde.

```
Gib eine Dezimalzahl ein: 6
Gib eine weitere Dezimalzahl ein: 0
Traceback (most recent call last):
  File "C:/Users/Kuhlmann/mu_code/0805_92_1.py", line 6, in
<module>
    c =a/b
ZeroDivisionError: float division by zero

Process finished with exit code 1
```

Die Division durch 0 wird except ValueError nicht abgefangen. Für einen solchen Fall muss eine weitere Exception-Regelung eingebaut werden.

```
except ZeroDivisionError:
    print("Division durch 0 ist nicht zulässig")
    antwort = input("Fehleingabe bitte bestätigen!")
```

Wird nun als zweite Zahl eine Null eingegeben, kommt es zu folgender Reaktion:

```
Gib eine Dezimalzahl ein: 6
Gib eine weitere Dezimalzahl ein: 0
Division durch 0 ist nicht zulässig
Fehleingabe bitte bestätigen!
```

Die try-except-Konstruktion ist nun eingekleidet in eine while-Schleife. Sie umschließt die Eingabe des zweiten Wertes sowie die beiden except-Blöcke.

Es werden hier zwei mögliche Fehleingaben abgefangen: unzulässiger Wert sowie 0 als Divisor. Wurde als zweiter Wert eine 0 eingegeben, meldet das System „Division durch 0 verboten!“ Auch die Eingabe eines Buchstabens anstelle einer Zahl führt jetzt nicht mehr zum Abbruch des Programms.

9.3 Exception in der GUI-Programmierung

Bei Gestaltung von Benutzerbildschirmen wird erwartet, dass sie benutzerfreundlich und fehlervermeidend aufgebaut sind. Bei der Programmierung ist zu überlegen, welche Fehler auftreten können, um Programmabstürze zu vermeiden.

Beispiel

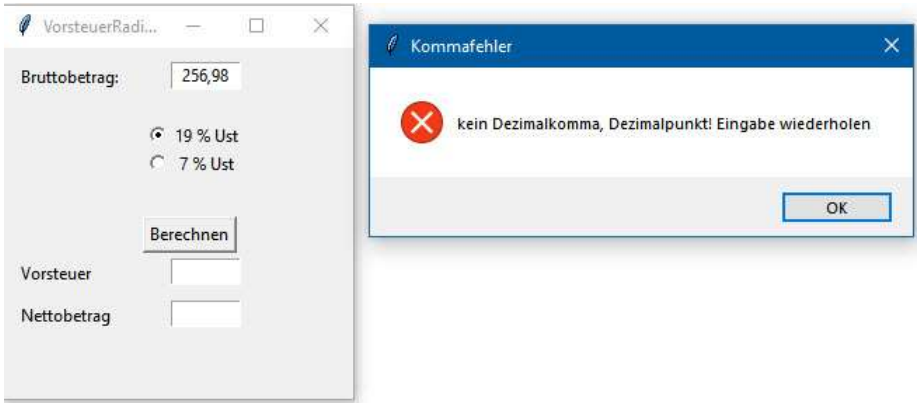
Das Programm, mit dem die Vorsteuer aus dem Bruttobetrag errechnet wird (s. vorheriger Abschnitt), hat eine Eingabeposition, in der per Tastatur ein Betrag eingegeben wird. Es ist nun zu überlegen, welche Fehler bei der Eingabe passieren können, die ein Programm abstürzen lassen. Die Eingabe eines falschen Betrages ist „ungefährlich“ und wird somit nicht per try-except-Konstruktion abgefangen. Fatal ist es jedoch, wenn eine Dezimalzahl mit Dezimalkomma oder ein Buchstabe eingegeben wird. Das führt zum Absturz. Somit muss für diesen Fall Vorsorge getroffen werden.


```
  File "C:\Program Files\Mu\pkgs\tkinter\__init__.py", line 1705, in __call__
    return self.func(*args)
  File "c:\users\gk\mu_code\radiobt_vorsteuer.py", line 41, in rechnung
    r_brutto=float(self.brutto.get())
ValueError: could not convert string to float: '256,98'
```

Weiterhin muss überlegt werden, welche Anweisung bzw. Anweisungen in einen try-Block gesetzt werden müssen. Das ist in diesem Fall recht einfach: Das System kann erst dann die Eingabe überprüfen, wenn die Schaltfläche *Berechnen* geklickt wurde. Also muss der try-Block unmittelbar vor der Anweisung r_brutto=float(self.brutto.get()) gesetzt werden.

Der except-Block könnte dann unmittelbar im Anschluss eingefügt werden. Die Einfügung würde aber eine Kette von zusammenhängenden Anweisungen unterbrechen. Deshalb wird der except-Block hier an das Ende der Funktion *Rechnen* gesetzt.

Der Nutzer des Programms muss auf die Fehleingabe hingewiesen werden. Das geschieht im except-Block. Dazu eignet sich am besten eine Messagebox.



Abschließend wird das Eingabefeld gelöscht, der Schreibcursor steht im Eingabefeld, der Nutzer kann mit der Eingabe beginnen.

```
    def rechnung(self):
        try:
            r_brutto=float(self.brutto.get())
            r_steuersatz = float(self.auswahl.get())
            r_netto= r_brutto/(1+r_steuersatz*0.01)
            r_steuer = format(r_brutto-r_netto, '.2f')
            self.ust.set(r_steuer)
            self.netto.set(value=format(r_netto,'.2f'))

        except ValueError:
            messagebox.showerror('Kommafehler','kein Dezimalkomma, '
                                               'Dezimalpunkt! Eingabe korrekt wiederholen!')
            self.brutto.set('  ')
```

Die try-except-Konstruktion kann auf bestimmte Ereignisse ausgerichtet werden.

Beispiele

KeyError: ereignet sich, wenn ein Schlüssel eines Dictionary nicht gefunden wird.

IndexError: Bei einer Listenoperation weist der Index über das Ende der Liste hinaus.

EOFError: wenn versucht wird, über den letzten Satz einer Datei hinaus zu lesen.

ImportError entsteht, wenn ein Import-Befehl das angegebene Modul nicht findet.

TypeError tritt auf, wenn eine Operation auf ein Objekt ausgeführt werden soll, die nicht vorgesehen ist, zum Beispiel der Versuch, die Wurzel-Funktion auf eine Liste anzuwenden.

Es können mehrere except-Klauseln in einer try-except-Konstruktion aufgeführt werden. Das except ohne weiteren Zusatz fängt alle Fehler ab. Es wird deshalb an das Ende der except-Auflistung gesetzt.

9.4 Zusammenfassung

- Programme müssen gegen Abstürze gesichert werden.
- Mittels einer try-except-Konstruktion kann ein Ereignis, das zu einem Programmabsturz führt, abgefangen werden.
- Im try-Anweisungsblock steht die Anweisung, bei der einer fataler Fehler auftreten kann. Im except-Block erfolgt die Fehlerbehandlung.
- Damit nach einer Fehlerbehandlung der Programmlauf fortgesetzt werden kann, wird die try-except-Konstruktion in eine While-Schleife eingebaut.
- Die try-except-Konstruktion kann auf bestimmte Ereignisse abgestimmt werden, wie zum Beispiel KeyError oder ValueError.

9.5 Aufgaben

Aufgabe 1

Ein Gleichungssystem der Form
a*x + b*y = c
d*x + e*y = f

kann man mithilfe der Determinantenformel lösen:

$$x = \frac{c*e - f*b}{a*e - d*b}$$

$$y = \frac{a*f - d*c}{a*e - d*b}$$

Schreiben Sie das Programm und berücksichtigen Sie dabei, dass eine Division durch 0 zu einem Programmabsturz führt.

Aufgabe 2

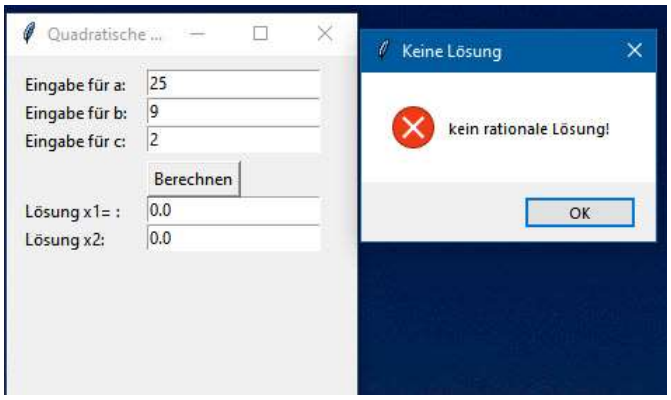
Quadratische Gleichungen der Form $ax^2 + bx + c = 0$ lassen sich nach der Formel

$$X_{1,2} = \frac{-b +- SQRT(b^2 - 4ac)}{2a}$$

lösen.

Entwickeln Sie eine Eingabemaske und programmieren Sie den Lösungsweg.

Beachten Sie, dass eine Gleichung auch nicht lösbar sein kann.



10 Datenbank

10.1 Einbindung einer Datenbank

Ein Datenbanksystem (DBS) speichert Daten in Tabellen und stellt Methoden zur Verfügung, mit denen man die Daten verwalten kann:

- Anlage einer Datenbank
- Anlage von Tabellen,
- Speichern von eingegebenen Daten
- Abrufen von Daten
- Löschen von Daten und Tabellen

Das Datenbanksystem verfügt aber nicht über die Methoden, die benötigt werden, um Daten in einer Bildschirmmaske zu verwalten. Dieser Teil der Datenbankanwendung wird Front-End genannt und wird mittels eines Programms gestaltet.

Damit von einem Anwenderprogramm auf eine Datenbank zugegriffen werden kann, wird ein weiteres Programm benötigt, das die Verbindung herstellt. Dieses Programm nennt man ODBC-Treiber. Der Begriff ODBC steht für Open Database Connectivity.

Im Folgenden wird die Datenbank sqlite3 eingesetzt. Sie ist eine kleine relationale Datenbank, die in Python integriert ist. Deshalb bedarf es keines weiteren Datenbankservers. Die Verwaltung der Datenbank erfolgt mittels SQL – Structured Query Language.

10.2 Arbeiten mit einer Datenbank

Die Einbindung von SQL in die Programmierung mit Python sei an einem kleinen Beispiel dargestellt. Unser Unternehmen betreibt eine LKW-Flotte. Bei der Erledigung von Transportaufträgen werden mautpflichtige Bundesstraßen und Autobahnen benutzt. Die Höhe der Mautgebühr hängt ab von der Schadensklasse des LKWs sowie von der Anzahl der Kilometer, die er auf mautpflichtigen Straßen zurückgelegt hat.

Mautgebühren

Schadensklasse	3B	4A	3D	4F	4E
Maut pro km	0,184	0,187	0,204	0,261	0,25

Die Ziffer gibt die Achs- und Gewichtsklasse des LKW an. Der Buchstabe steht für die Umweltbelastung.

Jeder LKW wird einer Schadensklasse zugeordnet.

Für die Abrechnung der Maut werden 3 Tabellen benötigt. In der Tabelle *lkw* wird das Kennzeichen des Lkws sowie seine Schadensklasse gespeichert. Die Tabelle *gebuehr* weist pro Schadensklasse die Mautgebühr pro Kilometer aus. In der Klasse *fahrtenbuch* werden pro LKW und Arbeitstag die gefahrenen mautpflichtigen Kilometer und der Fahrer vermerkt.

Tabelle	Datenfelder			
lkw	lkwkz		schadklasse	
gebuehr	schadklasse		euroKm	
fahrtenbuch	lkwKz	Datum	mautKm	Fahrernummer

Bevor die Tabellen in der Datenbank eingerichtet werden, ist zu prüfen, ob die Tabellen untereinander verknüpft werden können. Es ist also eine Join-Analyse durchzuführen.

Join-Analyse

lkw	gebuehr	fahrtenbuch
lkwkz		lkwlz
schadklasse	schadklasse	

Mittels der Attribute *lkwKz* und *schadklasse* können die Tabellen untereinander verknüpft werden.

10.2.1 Anlegen einer Datenbank

Bei Anwendung der Datenbank sqlite3 entfällt die Notwendigkeit, mittels der SQL-Anweisung CREATE DATABASE eine Datenbank anzulegen. Es genügt die Anweisung, das Modul sqlite3 zu importieren und eine Verbindung aufzubauen. Als Parameter der connect-Methode wird der Name der Datenbank angegeben, hier lkwMaut.db

```
#0805_10_1_daba00
import sqlite3
verbindung = sqlite3.connect("lkwMaut.db")
c=verbindung.cursor()
print('Datenbank angelegt!')
```

Der Zugriff auf die Datenbank wird mittels eines Cursors gesteuert. Die Objektvariable für den Cursor() wurde hier kurzerhand **c** genannt.

Die oberen 4 Anweisungen sind in jedem Python-Programm zu formulieren, mit auf die interne Datenbank zugegriffen wird.

10.2.2 Anlegen einer Tabelle

Die Anweisung zum Aufbau einer Tabelle lautet in SQL

CREATE TABLE tabellename()

Hinter dem Tabellennamen werden in runden Klammern der Name der Spalten sowie ihr Datentyp angegeben.

```
#0805_10_23_daba01
import sqlite3
verbindung = sqlite3.connect("lkwmaut.db")
c=verbindung.cursor()
c.execute("""CREATE TABLE lkw (lkwKz TEXT(10),schadklasse TEXT (2))""")
verbindung.commit()
```

Die Tabelle lkw weist zwei Attribute auf: *lkwKz* und *schadklasse*. Ihr Datentyp ist TEXT mit der Angabe der benötigten Stellenanzahl.

Hinweis: Es ist üblich, die SQL-Anweisungen in Großbuchstaben zu schreiben. Es dient aber nur der Übersichtlichkeit. SQL versteht die Anweisungen auch in Kleinbuchstaben.

Die SQL-Anweisung steht zwischen Anführungszeichen. Erstreckt sich die Anweisung über mehr als eine Zeile, werden sie zwischen drei Anführungszeichen gesetzt:

```
cursorvariable.execute(“““SQL-Anweisung“““)
```

Datentypen

Sqlite3 kennt folgende Datentypen:

Datentyp	Sqlite 3	Python
Ganzzahl	INTEGER	Int
Dezimalzahl	REAL	Float
Zeichen / Zeichenfolge	TEXT()	Str
Byte	BLOB	bytes
Nichts	NULL	None

10.2.3 Daten in eine Tabelle schreiben

Die SQL-Anweisung, mit der Daten in eine Tabelle geschrieben werden, lautet:

```
INSERT INTO tabellenname VALUES()
```

Beispiel

```
INSERT INTO lkw VALUES ("K-PY666","4A")
```

In die Spalte *lkwKz* wird *K-PY666* und in Spalte *schadklasse 4A* eingetragen.

Sind viele Daten einzugeben, kann man sich die Schreibarbeit erleichtern. Der Dateneintrag erfolgt in einer for-Anweisung, wie das nachstehende Beispiel zeigt.

```
#0805_1022_daba01

for lkwdaten in (
        ("K-PY666","4A"),
        ("K-PY111","3B"),
        ("K-PY222","4C"),
        ("K-PY333","3D"),
        ("K-PY444","4F"),
        ("K-PY555","4E")):
   c.execute("INSERT INTO lkw VALUES (?,?)",lkwdaten)
verbindung.commit()
```

In der for-Anweisung werden der Reihe nach die LKW-Daten an das INSERT-INTO-Statement gegeben. Die zwei Fragenzeichen in der VALUES-Klausel geben an, dass zwei Daten aus dem Listenelement von *lkwdaten* in das SQL-Statement übernommen werden.

Die Tabelle gebuehr wird mit folgendem Programm aufgebaut:

```
#0805_1023_daba02
import sqlite3
verbindung = sqlite3.connect("lkwmaut.db")
c=verbindung.cursor()
c.execute("""CREATE TABLE gebuehr (schadklasse TEXT (2),eurokm REAL)""")
for mautdaten in (
        ("4A",0.187),
        ("3B",0.184),
        ("4C",0.208),
        ("3D",0.204),
        ("4F",0.261),
        ("4E",0.25)):
```

```
   c.execute("INSERT INTO gebuehr VALUES (?,?)",mautdaten)
                verbindung.commit()
```

Für den Aufbau der Tabelle *fahrtenbuch* lauten die Anweisungen:

```
#0805_1023_daba03 Fahrtenbuch
import sqlite3
verbindung = sqlite3.connect("lkwmaut.db")
c=verbindung.cursor()
c.execute("""CREATE TABLE IF NOT EXISTS fahrtenbuch (lkwkz TEXT(10),
            datum date, mautkm INTEGER, fahrer TEXT(3))""")
for fahrdaten in (
          ("K-PY555","01.10.2022",202,"101"),
          ("K-PY111","02.10.2022",362,"102"),
          ("K-PY222","04.10.2022",125,"101"),
          ("K-PY333","02.10.2022", 391,"103"),
          ("K-PY666","04.10.2022",254,"103"),
          ("K-PY444","05.10.2022",295,"102")):
   c.execute("INSERT INTO fahrtenbuch VALUES (?,?,?,?)",fahrdaten)
verbindung.commit()
c.close()
```

Der Fahrer wird durch eine Personalnummer repräsentiert. Die Fahrertabelle wird später eingeführt.

In dieser CREATE-Anweisung wurde ein IF-Klausel verwendet. Dadurch wird verhindert, dass das Programm abstürzt, wenn die Tabelle schon besteht. Wird diese Anweisung erneut aufgerufen, werden die die Angaben in *fahrdaten* erneut in die Tabelle eingetragen.

10.2.4 Auslesen von Daten aus einer Tabelle

In dem folgenden Programm werden die Daten der Tabelle *fahrtenbuch* ausgelesen.

```
#0805_1024_Allabfrage
import sqlite3
verbindung = sqlite3.connect("lkwMaut.db")
c=verbindung.cursor()

c.execute(""" SELECT * FROM fahrtenbuch""")
for zeile in c:
   print(zeile)

for zeile in c:
   a= list(zeile)
   print(a[0],a[1],a[2],a[3])
```

```
    print(a[1], a[0], a[2] )
c.close
```

Um die Daten aus einer Tabelle auszulesen, wird das SELECT-Statement eingesetzt. Sind alle Daten einer Tabelle auszugeben, lautet das Statement

```
SELECT * FROM tabelle
```

Durch die Angabe des Zeichens * hinter SELECT werden alle Spalten einer Tabelle angesprochen. Sollen die Daten einzelner Spalten ausgelesen werden, sind die Namen der entsprechenden Spalten hinter SELECT aufzuführen. Die SELECT-Anweisung kann vielfältig ausgestaltet werden. Darauf wird in den folgenden Abschnitten eingegangen.

```
c.execute("SELECT * FROM fahrtenbuch")
```

Das Ergebnis einer SELECT-Abfrage wird als Tupel ausgegeben. In einer for-Schleife

```
for zeile in c::
   print(zeile)
```

werden die Daten auf dem Bildschirm angezeigt:

```
0805_1024_AllAbfrag
('K-PY555', '01.10.2022', 202, '101')
('K-PY111', '02.10.2022', 362, '102')
('K-PY222', '04.10.2022', 125, '101')
('K-PY333', '02.10.2022', 391, '103')
('K-PY666', '04.10.2022', 254, '103')
('K-PY444', '05.10.2022', 295, '102')
```

Möchten Sie die Daten einzeln ausgeben oder einzeln verarbeiten, übergeben Sie die Daten in eine Liste:

```
a = list(zeile).
```

Der Zugriff auf die Daten erfolgt dann durch Listenname[index]:

```
for zeile in c:
    a = list(zeile)
    print(a[0], a[1],a[2], a[3])
```

Ausgabe:

```
K-PY555 01.10.2022 202 101
K-PY111 02.10.2022 362 102
K-PY222 04.10.2022 125 101
K-PY333 02.10.2022 391 103
K-PY666 04.10.2022 254 103
K-PY444 05.10.2022 295 102
```

10.2.5 Datenbank schließen

Wurde die Datenbank in dieser Weise geöffnet

```
verbindung = sqlite3.connect("lkwmaut.db")
c=verbindung.cursor()
```

wird sie durch

```
c.close
```

geschlossen.

10.2.6 Eine Tabelle löschen

Das SQL-Statement löscht alle gerade angelegten Tabellen:

```
import sqlite3
verbindung = sqlite3.connect("lkwmaut.db")
c=verbindung.cursor()
c.execute("""DROP TABLE gebuehr""")
c.execute("""DROP TABLE lkw""")
c.execute("""DROP TABLE fahrtenbuch""")
c.close()
```

Mittels

```
DROP TABLE tabellenname
```

wird eine Tabelle gelöscht.

10.2.7 Tabellenzeilen unter einer Bedingung auswählen

Einfache Bedingung

Beispiel: Man möchte wissen, welche LKWs mehr als 300 km zurückgelegt haben.

Bei dieser Problemstellung sollen nur die Fahrzeuge angezeigt werden, bei denen die Bedingung erfüllt ist. LKWs mit geringerer Kilometerzahl sind nicht von Interesse, sollen also nicht angezeigt werden.

Für die Zeilenauswahl wird eine Bedingung benötigt. Sie wird in SQL in einer WHERE-Klausel formuliert:

```
WHERE spaltenname Vergleichsoperator Vergleichswert
```

Folgendes Programm löst die gestellte Aufgabe:

```
#08050_1027_daba04
import sqlite3
verbindung = sqlite3.connect("lkwmaut.db")
c=verbindung.cursor()
c.execute("""SELECT lkwkz, mautkm FROM fahrtenbuch WHERE mautkm>300""")
```

```
for zeile in c:
    print(zeile)
c.close()
```

Ausgabe:

```
('K-PY111', 362)
('K-PY333', 391)
```

Wie an der Ausgabe zu sehen ist, werden nur die LKWs angezeigt, die mehr als 300 km gefahren sind.

Vergleichsoperatoren in SQL:

Operator	Bedeutung
=	gleich
<	kleiner als
>	größer als
<>	ungleich
<=	kleiner oder gleich
>=	größer oder gleich
LIKE	Mustervergleich
BETWEEN	Bereichsabfrage

In dieser Abfrage wurden auch nicht alle Spalten der Tabelle *fahrtenbuch* angezeigt. Es wurden nur die angesprochen, die hinter SELECT angegeben waren.

Prepared Statement

Wird in der WHERE-Klausel auf eine Zeichenkette abgefragt, die in einer Variablen gespeichert ist, so muss ein prepared statement verwendet werden.

Beispiel

An welchen Tagen wurde ein bestimmter LKW eingesetzt?

```
#0805_1027_EinsatzLKW
import sqlite3
autonr = 'K-PY444'
verbindung = sqlite3.connect("lkwmaut.db")
c=verbindung.cursor()
c.execute("select lkwKz, datum from fahrtenbuch where lkwKz=?",(autonr,))
for zeile in c:
    print(zeile)
```

```
c.close()
```

In der WHERE-Klausel wird die Vergleichervariable nicht direkt angegeben. Dafür steht ein Fragezeichen. Wofür das Fragezeichen steht, wird im Anschluss an das SQL-Statement in einer runden Klammer angegeben. Das letzte Zeichen in der Tupel-Klammer muss ein Komma sein.

Der Grund für diese Vorgehensweise hängt mit den verschiedenen internen Übersetzungs- und Verbindungsprozessen zusammen.

Zusammengesetzte Bedingung

Beispiel**:** Welche LKWs sind nach dem 2. Oktober 2022 weniger als 300 km gefahren?

```
#0805_1027_datum_daba04
import sqlite3
verbindung = sqlite3.connect("lkwmaut.db")
c=verbindung.cursor()
c.execute("""SELECT lkwKz, datum, mautkm FROM fahrtenbuch
                WHERE mautkm < 300 AND datum > '02.10.2022' """)
for zeile in c:
    print(zeile)

c.close()
```

Ergebnis:

```
0805_1027_datum_daba04
('K-PY222', '04.10.2022', 125)
('K-PY666', '04.10.2022', 254)
('K-PY444', '05.10.2022', 295)
```

Hier wurden zwei Bedingungen mittels AND verknüpft. Es werden nur die Datenzeilen angezeigt, in denen beide Bedingungen erfüllt sind.

SQL kennt außer der UND-Verknüpfung die ODER-Verknüpfung OR sowie die Verneinung NOT.

Das obige SELECT-Statement enthält zwei Vergleichswerte, die sich in ihrem Datentyp unterscheiden. Es ist zu beachten, dass ein Vergleichswert, der nicht numerisch ist, in Hochkomma gesetzt werden muss.

Bedingung mit Mustervergleich

Die Fähigkeit, Daten über einen Mustervergleich aus einer Tabelle herauszusuchen, ist ein wichtiges Leistungsmerkmal einer Datenbank.

Beispiel: Die Schadensklasse setzt sich aus zwei Zeichen zusammen. Das erste Zeichen, die Ziffer, gibt die Gewichtsklasse an. Das zweite Zeichen steht für den Grad der Umweltbelastung.

Welche Schadensklasse haben die LKWs der Gewichtsklasse 4.

```
#0805_1027_LIKE
import sqlite3
verbindung = sqlite3.connect("lkwmaut.db")
c=verbindung.cursor()
#--- Like ----------------
c.execute("""SELECT lkwkz, schadklasse FROM lkw WHERE schadklasse LIKE '4%' """)
for zeile in c:
    print(zeile)
```

Ergebnis:

```
('K-PY666', '4A')
('K-PY222', '4C')
('K-PY444', '4F')
('K-PY555', '4E')
```

Bei dieser Abfrage kann nicht auf Gleichheit gefragt werden, da nur ein Teil des Vergleichsbegriffs ausgewertet werden soll – eben nur das erste Zeichen. Die übrigen Teile des Vergleichsbegriffs sind für die Auswahl der Zeilen nicht wichtig. Hier wird also auf ein Muster abgefragt. Dafür wird der Vergleichsoperator LIKE eingesetzt.

Für den Teil des Vergleichsbegriffs, der nicht wichtig für die Auswahl nicht wichtig ist, wird ein Platzhalter eingesetzt. SQL unterscheidet folgende Platzhalter bzw. Joker:

Platzhalter	Bedeutung
%	Steht für eine beliebige Anzahl beliebiger Zeichen
_ (Unterstrich)	Steht für ein Zeichen

In dem obigen Beispiel hätte man auch wie folgt formulieren können:

```
WHERE schadklasse LIKE '4_'
```

Beispiele für Mustervergleiche

Die Spalte Name enthält die Hausnamen: Maier, Mayer, Meier, Myer, Meyer, Meieris

WHERE name LIKE '_y%' → Myer

WHERE name LIKE ‘%y%‘ → Mayer, Myer, Meyer

WHERE name LIKE ‘%ei%‘ → Meier, Meieris

WHERE name LIKE ‘%ris‘ → Meieris

WHERE name LIKW ‘_a%eris‘ →keine Ausgabe

Bedingung mit Bereichsangabe

Beispiel**:** Welche Fahrzeuge waren zwischen dem 01.10.2022 und dem 03.10.2020 im Einsatz?

```
#0805_Between
import sqlite3
verbindung = sqlite3.connect("lkwmaut.db")
c=verbindung.cursor()
c.execute("""SELECT lkwkz, datum FROM fahrtenbuch
                WHERE datum BETWEEN '01.10.2022' AND '03.10.2022'""")
for zeile in c:
   print(zeile)
c.close
```

Ergebnis:

```
0805_1027_Between
('K-PY555', '01.10.2022')
('K-PY111', '02.10.2022')
('K-PY333', '02.10.2022')
```

Die Bereichsabfrage wird mittels des Operators BETWEEN durchgeführt. Er kann sowohl bei numerischen Spalten wie auch bei Textspalten und Datumsspalten eingesetzt werden. BETWEEN wählt die Daten aus, die sich zwischen dem unteren und dem oberen Bereichswert befinden. Die beiden Bereichswerte werden mit einbezogen.

10.2.8 Aggregatfunktionen

Für das Fahrzeugmanagement ist die Beantwortung folgender Fragen wichtig:

- Wie viele Kilometer sind die Fahrzeuge in dieser Woche gefahren?
- Welches Fahrzeug hat die meisten Kilometer zurückgelegt?
- Wie hoch ist die durchschnittliche Fahrleistung der LKWs gewesen?

Für die Beantwortung dieser Fragen werden die Aggregatfunktionen eingesetzt. Diese Funktionen werden auch Spaltenfunktionen genannt. Sie ermitteln für eine einzelne Tabellenspalte aus einer Gesamtmenge die Summe,

den Minimal- oder Maximalwert, den Mittelwert oder die Anzahl der Zeilen einer Tabelle.

Funktion	Ergebnis
SUM(spaltenname)	Summe der numerischen Werte in einer Spalte
AVG(spaltenname)	Durchschnitt der numerischen Werte einer Spalte
MIN(spaltenname)	Kleinster Wert in einer Spalte
MAX(spaltenname)	Größter Wert in einer Spalte
COUNT(*)	Anzahl der vorhandenen Zeilen einer Tabelle
COUNT(DISTINCT spaltenname)	Anzahl der Zeilen mit unterschiedlichen Werten

Beispiel: Wie viele Kilometer haben die Fahrzeuge zurückgelegt?

Wie hoch war die durchschnittliche Kilometerleistung?

```
#0805_1028_Aggregat
import sqlite3
verbindung = sqlite3.connect("lkwmaut.db")
c=verbindung.cursor()
c.execute("""SELECT SUM(mautkm) FROM fahrtenbuch """)
print('Summe der gefahrenen Kilometer')
for zeile in c:
    print(zeile)
c.execute("""SELECT AVG(mautkm) FROM fahrtenbuch """)
print('Durchschnittliche Kilometerleistung')
for zeile in c:
    print(zeile)
c.close(
```

```
Summe der gefahrenen Kilometer
(1629,)
Durchschnittliche Kilometerleistung
(271.5,)
```

Ergebnis:

10.2.9 Daten sortieren und gruppieren

Die Auswertung von Daten wird leichter, wenn sie sortiert oder in Gruppen zusammengefasst werden. Die sortierte Ausgabe von Daten bewirkt der Zusatz ORDER BY.

Beispiel: Die Kennzeichen der LKWs sollen alphabetisch aufgelistet werden.

```
#0805_1029_Sortieren
import sqlite3
verbindung = sqlite3.connect("lkwmaut.db")
c=verbindung.cursor()
c.execute("""SELECT lkwkz FROM fahrtenbuch ORDER BY lkwkz""")
for zeile in c:
    print(zeile)
```

Ergebnis:

```
('K-PY111',)
('K-PY222',)
('K-PY333',)
('K-PY444',)
('K-PY555',)
('K-PY666',)
```

Für das Gruppieren wird die SELECT-Anweisung um die Klausel GROUP BY erweitert.

Beispiel: Man möchte wissen, wie viele Fahrzeuge an den verschiedenen Tagen unterwegs waren.

```
#0805_1029_Gruppieren
import sqlite3
verbindung = sqlite3.connect("lkwMaut.db")
c=verbindung.cursor()
c.execute("""SELECT  datum, count(*) FROM fahrtenbuch GROUP BY datum """)
for zeile in c:
```

```
    print(zeile)
close()
```

Ergebnis:

```
('01.10.2022', 1)
('02.10.2022', 2)
('04.10.2022', 2)
('05.10.2022', 1)
```

Das Datum ist der Gruppierungsbegriff. Über die Datumsangabe wird eine Gruppe gebildet und gezählt, wie viele Elemente die einzelnen Gruppen aufweisen. Am 3. Oktober war kein Fahrzeug unterwegs, da dieser Tag in Deutschland Nationalfeiertag ist.

Der GROUP- BY-Zusatz steht stets am Ende eines SELECT-Statements.

10.2.10 Ergebnis einer Abfrage in eine Tabelle schreiben

Problem

Die Mautgebühren sind zu berechnen. Dazu wird eine Tabelle eingerichtet, in der pro Fahrzeug die Summe der gefahrenen Kilometer gespeichert werden kann.

```
0805_10210_InsertInto
c.execute("""CREATE TABLE IF NOT EXISTS zwidat3
                                        (lkwkz TEXT(10), sumkm INTEGER)""")
```

In diese Zwischentabelle wird pro Fahrzeug die Summe der gefahrenen Kilometer geschrieben. Das geschieht durch ein INSERT-Statement:

```
c.execute("""INSERT INTO zwidat3 (lkwkz, sumkm)
        SELECT lkwkz, sum(mautkm) FROM fahrtenbuch GROUP BY lkwkz""")
```

Inhalt von zwidat3:

```
0805_1029_InsertInto
('K-PY111', 362)
('K-PY222', 125)
('K-PY333', 391)
('K-PY444', 295)
('K-PY555', 202)
('K-PY666', 254)
```

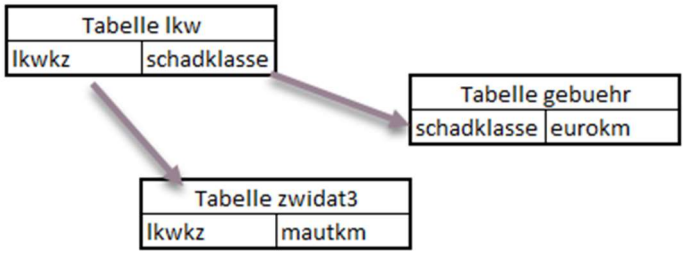
In diesem Beispiel wird das Ergebnis der SELECT-Abfrage in die Tabelle *zwidat3* geschrieben. Die Summenfunktion bewirkt in Verbindung mit dem GROUP-BY-Zusatz, dass pro Fahrzeug die Summe der gefahrenen Kilometer berechnet.

10.2.11 Verknüpfung von Tabellen

Beispiel: Die Mautgebühren sind für jeden LKW zu berechnen.

Dazu werden zwei weitere Tabellen benötigt. In der Tabelle *lkw* ist die Schadensklasse für jeden LKW angegeben. In der Tabelle *gebuehr* ist für jede Schadensklasse die Kilometergebühr eingetragen.

Join-Analyse



Die Tabelle *lkw* ist über *lkwkz* mit der Tabelle zwidat3 verbunden. Zur Tabelle *gebuehr* stellt das Attribut *schadklasse* die Verbindung her.

Die Verbindung zwischen Tabellen kann mit der Join-Klausel, aber auch mittels der WHERE-Klausel hergestellt werden.

```
WHERE lkw.lkwkz = zwidat3.lkwkz AND lkw.schadklasse = gebuehr.schadklasse
```

Um die Gebühr zu ermitteln, sind *mautkm* und *eurokm* miteinander zu multiplizieren:

sumkm * eurokm

Die SQL-Anweisung zur Berechnung der Mautgebühr pro Fahrzeug und Kilometerleistung lautet dann:

```
#0805_102211_Verknüpfung
...
c.execute("""SELECT lkw.lkwkz, zwidat3.sumkm * gebuehr.eurokm
                FROM zwidat3, lkw, gebuehr
                WHERE zwidat3.lkwkz = lkw.lkwkz AND
                        lkw.schadklasse = gebuehr.schadklasse""")
```

Ergebnis:

```
gebuehren ----
('K-PY111', 66.608)
('K-PY222', 26.0)
('K-PY333', 79.764)
('K-PY444', 76.995)
('K-PY555', 50.5)
('K-PY666', 47.498)
```

Qualifizierte Bezeichner

In dem obigen Beispiel wurden für verschiedene Tabellen gleichlautenden Spaltenbezeichner verwandt. Damit das Datenbanksystem weiß, aus welcher Tabelle die Werte zu nehmen sind, wird der Tabellenname vor den Spaltennamen gesetzt: tabellenname.spaltenname. Diese Form wird qualifizierter Bezeichner genannt.

Rechenoperationen

Soll die Mautgebühr in einer Summe ausgewiesen werden, ist das SQL-Statement leicht zu verändern:

```
#Fortsetzung 0805_10211_Verknüpfung

c.execute("""SELECT sum(zwidat3.sumkm*gebuehr.eurokm)
                FROM zwidat3, lkw, gebuehr
                 WHERE zwidat3.lkwkz = lkw.lkwkz AND
                lkw.schadklasse = gebuehr.schadklasse""")
```

Ausgegeben wird nun die Gebührensumme: 347, 36

Dieses Beispiel zeigt, dass in einem SQL-Statement auch Rechenoperationen durchgeführt werden. Die Rechenoperatoren stimmen mit denen von Python überein.

Auch mit den Datumsangaben kann man rechnen:

```
c.execute("""SELECT '15.10.2022' - datum
                 FROM fahrtenbuch
                WHERE lkwkz = 'K-PY555'""")
```

Ergebnis: 14

Beim LKW K-PY555 war in der Datumsspalte 01.10.2022 eintragen. Zwischen dem 1. Oktober und dem 15. Oktober liegen 14 Tage.

10.2.12 Zusammenfassung

- Eine Datenbank besteht aus mehreren Tabellen, die untereinander verknüpft werden können.
- Um die Möglichkeit der Verknüpfung zu überprüfen, ist eine Join-Analyse vorzunehmen.

- Mittels con = sqlite3.connect(“Datenbanknamen.db“) wird die Datenbank unter dem angegebenen Namen angelegt.
- Der Zugriff auf die Datenbank erfolgt durch eine Cursor: zeiger = con.cursor().
- Die SQL-Anweisungen wird in Form eines Parameters der Methode zeiger.execute(“““ sql-Statement “““) formuliert.
- con.commit stellt sicher, dass die Änderungen in der Datenbank gespeichert werden.
- Sqlite3 kennt die Datentypen INTEGER; REAL und TEXT().
- CREATE TABLE IF NOT EXISTS name(Attibut Datentyp, …) generiert eine Tabelle, sofern sie noch nicht existiert.
- INSERT INTO tabellenname VALUES () trägt Daten in eine Tabelle ein.
- SELECT * FROM tabelle zeigt den Inhalt einer Tabelle an.
- Zeiger.close() schließt die Datenbank.
- In der WHERE-Klausel werden Auswahlbedingungen formuliert.
- Im Mustervergleich LIKE ist der Unterstrich Platzhalter für ein einzelnes Zeichen, % steht für beliebig viele Zeichen.
- Aggregatfunktionen liefern einen komprimierten Wert pro Spalte.
- Eine sortierte Ausgabe bewirkt die ORDER BY-Klausel.
- Zur gruppenweisen Auswertung wird die GROUP-BY-Klausel benötigt.

10.2.13 Aufgaben

Aufgabe 1

Richten Sie in der Datenbank *lkwmaut.db* die Tabelle *trucker* mit den Attributen ein:

Tabelle trucker		
fahrer	name	Fixum in Euro
101	Emre	42,00
102	Berta	41,00
103	Alberto	43,00

Aufgabe 2

Die Fahrerliste soll in alphabetischer Ordnung angezeigt werden.

Aufgabe 3

Wann war der Fahrer mit der Personalnummer 101 im Einsatz.

Aufgabe 4

Mit welchem Fahrzeug (lkwkz) war der Fahrer 103 am 02.10.2022 unterwegs

Aufgabe 5

Wie hoch ist das durchschnittliche Fixum, das die Firma an die Fahrer zahlt?

Aufgabe 6

Wie oft war der Fahrer 102 im Einsatz?

Aufgabe 7

Welche Fahrer waren am 02.10.2022 mit welchen Fahrzeugen unterwegs?

Aufgabe 8

Die Fahrer erhalten neben ihrem Fixum pro Tag ein Kilometergeld. Das beträgt pro Kilometer 0,20 €. Wie hoch ist der Verdienst des Fahrers mit der Personalnummer 101 am 01.10.2022

Aufgabe 9

Wie hoch ist die Lohnsumme, die die Firma in dem Zeitraum 01. – 05.Oktober an die Fahrer zahlen muss.

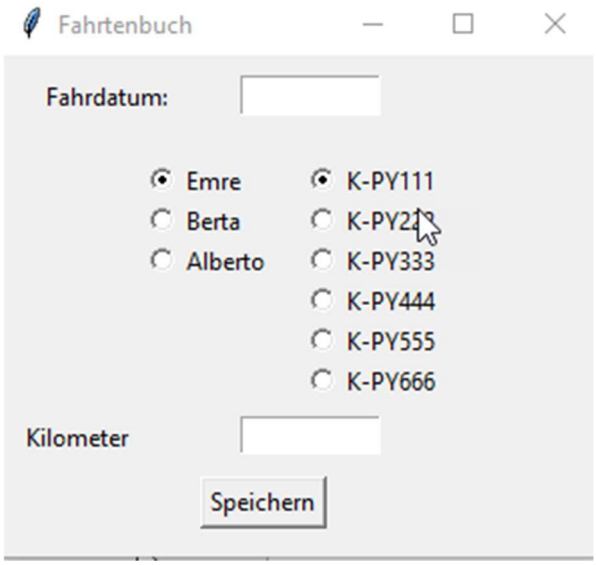
10.3 Grafische Bedienoberfläche und Datenbankzugriff

10.3.1 Maskenaufbau

Problem

Für jeden Arbeitstag wird festgehalten, welcher Fahrer mit welchem Fahrzeug wie viele mautpflichtige Kilometer gefahren ist. Die Daten werden in der Tabelle *fahrtenbuch* gespeichert. Die Bildschirmmaske soll in der Weise gestaltet sein, dass die Eingabe der erforderlichen Daten rasch erfolgen kann.

Lösung



Bei dieser Problemstellung bieten sich Radiobutton an, mit denen der Fahrer und das Fahrzeug angegeben werden. Als Datum könnte man auch das Systemdatum nehmen. Da aber die Dateneingabe erst einige Tage nach dem Fahrdatum erfolgt, muss das Datum bei der Erfassung eingetippt werden.

Programm

```
# 0805_1031 erfassenFahrtenbuch
from tkinter import *

class erfassenFahrtenbuch:

    def __init__(self):

        self.fenster = Tk()
        self.datum = StringVar()
```

```
        self.mautkm = StringVar()
        self.fahrer = StringVar()
        self.fahrer.set('101')
        self.auswahl = StringVar()
        self.auswahl.set('K-PY111')
        self.fenster.title('Fahrtenbuch')
        self.fenster.geometry('300x250+700+350')

        lab1=Label(self.fenster,text='Fahrdatum:')
        lab1.place(x=20, y=10)
        e1=Entry(self.fenster,textvariable=self.datum)
        e1.place(x=120, y=10, width=70,height=20)
        rb01 = Radiobutton(self.fenster, text='Emre', variable=self.fahrer, value='101')
        rb01.place(x=70, y=50)
        rb02 = Radiobutton(self.fenster, text='Berta', variable=self.fahrer, value='102')
        rb02.place(x=70, y=70)
        rb03 = Radiobutton(self.fenster, text='Alberto', variable=self.fahrer,
                           value='103')
        rb03.place(x=70, y=90)

        rb1=Radiobutton(self.fenster,text='K-PY111', variable=self.auswahl,
                        value='K-PY111')
        rb1.place(x=150, y = 50)
        rb2=Radiobutton(self.fenster, text='K-PY222', variable=self.auswahl,
                        value='K-PY222')
        rb2.place(x=150, y=70)
        rb3=Radiobutton(self.fenster, text='K-PY333', variable=self.auswahl,
                        value='K-PY333')
        rb3.place(x=150, y=90)
        rb3 = Radiobutton(self.fenster, text='K-PY444', variable=self.auswahl,
                          value='K-PY444')
        rb3.place(x=150, y=110)
        rb3 = Radiobutton(self.fenster, text='K-PY555', variable=self.auswahl,
                          value='K-PY555')
        rb3.place(x=150, y=130)
        rb3 = Radiobutton(self.fenster, text='K-PY666', variable=self.auswahl,
                          value='K-PY666')
        rb3.place(x=150, y=150)
        lab4 = Label(self.fenster, text='Kilometer')
        lab4.place(x=10, y=180)
        e3 = Entry(self.fenster, textvariable=self.mautkm)
        e3.place(x=120, y=180, width=70)
        bt=Button(self.fenster, text='Speichern',command=self.speichern)
        bt.place(x=100,y=210)
```

```
        self.fenster.mainloop()

    def speichern(self):
        db_datum = self.datum.get()                                          #1
        db_mautkm = self.mautkm.get()
        db_fahrer= self.fahrer.get()
        db_lkw= self.auswahl.get()

        import sqlite3                                                       #2
        verbindung = sqlite3.connect("lkwmautAufgaben.db")
        c=verbindung.cursor()
        c.execute("INSERT INTO fahrtenbuch VALUES (?,?,?,?)",
                                (db_lkw, db_datum, db_mautkm, db_fahrer)     #3
        verbindung.commit()
        print("Eintrag erfolgt")

        c.execute("""Select * FROM fahrtenbuch""")                           #4
        for zeile in c:
            print(zeile)
        c.close()

fb = erfassenFahrtenbuch()
```

Erläuterungen

#1 Die Eingabedaten werden aus der Bildschirmmaske ausgelesen.

#2 Das Datenbankmodul wird importiert und die Verbindung zur Datenbank *lkwmautAufgaben.db* aufgebaut.

#3 Die Eingabedaten werden in die Tabelle *fahrtenbuch* eingetragen. Es ist darauf zu achten, dass die Namen der Variablen, die die Eingabedaten enthalten, in einer runden Klammer stehen.

#4 Zur Kontrolle werden die Daten ausgelesen

10.3.2 Zusammenfassung

- Bei der Auswahl der Bedienelemente ist die Art und der Umfang der Datenvorgaben zu berücksichtigen.
- Das Bildschirmprogramm wird als Klasse ausgelegt.
- Die ausführenden Funktionen werden außerhalb der Bildschirmklasse platziert.

- Vor der Speicherung wird die Datenbank geöffnet, nach der Speicherung wird sie geschlossen. Dadurch wird sichergestellt, dass die Daten in der Datenbank gespeichert sind.

10.3.3 Aufgaben

Aufgabe 1

Ein neuer Fahrer wurde eingestellt. Sein Name ist Viktor. Er hat die Fahrernummer 104. Sein Tagessatz beträgt 42,00 Euro. Erweitern Sie die Tabelle trucker in der Datenbank lkwMautAufgaben.db um diese Daten.

Aufgabe 2

Passen Sie die Bildschirmmaske „Fahrtenbuch“ der geänderten Personalsituation an. Geben Sie mittels der Bildschirmmaske einige Fahrdaten des neuen Fahrers für den Zeitraum 01.10. bis 05.10.2022

Aufgabe 3

Die Firma braucht ein Programm für die Kalkulation von Transportaufträgen.

Personalkosten: pro Kilometer 0,90 Euro

Fahrerpauschale:

- Bis 100 km → 10,00 Euro
- Bis 200 km → 25,00 Euro
- Ab 201 km → 40,00 Euro

Kraftstoff in Abhängigkeit der Belastungsklasse des eingesetzten Fahrzeugs.

- Bei der Belastungsklasse 3 beträgt der Verbrauch 30 Liter auf 100 km.
- Bei der Belastungsklasse 4 sind für den Verbrauch 40 Liter auf 100 km anzusetzen.
- Der Preis für den Kraftstoff wird mit 1.50 Euro angesetzt.

Gemeinkostenzuschlag:

- Belastungsklasse 3: 40 % der Kraftstoffkosten
- Belastungsklasse 4: 50% der Kraftstoffkosten

In der Bildschirmmaske ist die Kilometerzahl sowie der LKW anzugeben.

Auszugeben ist der Angebotspreis.

11 Anhang

11.1 Einsatz von tkinter im Anfangsunterricht

Von Lehrkräften wird die Ansicht vertreten, dass die SuS stärker motiviert sind, wenn sie auf der Grundlage einer GUI das Programmieren lernen.

Das ist möglich, indem Vorlagen benutzt werden. In den Vorlagen sind die Labels und Textfelder der Aufgabenstellung entsprechend zu benennen. Die eigentliche Programmierung erfolgt dann in den Methoden.

Beispiel

```
from tkinter import *
from math import *
class Fenstervordruck:

    def __init__(self):

        self.fenster = Tk()
        self.ein1 = DoubleVar()
        self.ein2 = DoubleVar()
        self.aus1 = DoubleVar()
        self.aus2 = DoubleVar()
        self.fenster.title('Fenstertitel')
        self.fenster.geometry('300x200+100+150')

        l1=Label(self.fenster,text = 'Eingabe Wert1 ')
        l1.place(x=10,y=10)
        l2=Label(self.fenster,text = 'Eingabe Wert2')
        l2.place(x=10, y=30)
        e1=Entry(self.fenster,textvariable = self.ein1)
        e1.place(x=120,y=10)
        e2 = Entry(self.fenster, textvariable=self.ein2)
        e2.place(x=120, y=30)

        bt=Button(self.fenster, text='Berechnen',command=self.rechnung)
        bt.place(x=110,y=60)
        l3=Label(self.fenster,text='Ausgabe Wert1')
        l3.place(x=10,y=90)
        e3=Entry(self.fenster,textvariable=self.aus1)
        e3.place(x=120,y=90)
        l4=Label(self.fenster, text='Ausgabe Wert2')
        l4.place(x=10, y=130)
```

```
        e4=Entry(textvariable = self.aus2)
        e4.place(x=120,y=130)

        self.fenster.mainloop()

    def rechnung(self):
        var1=self.ein1.get()
        var2=self.ein2.get()

fv = Fenstervordruck()
```

Mit dieser Vorlage kann eine Problemstellung bearbeitet werden, bei der zwei Daten einzugeben sind sowie zwei Ergebnisse anzuzeigen sind. Die Ermittlung der Ergebnisse erfolgt in der Methode rechnung(self)

Um es noch weiter zu vereinfachen, kann die Bearbeitung der GUI im Unterrichtsgespräch erfolgen, so dass bei allen SuS die gleiche Ausgangslage besteht.

Verbindung mit Excel

Python verfügt über ein Modul, mit dem CSV-Dateien, die in Excel erzeugt worden sind, einlesen und auch wieder zurückschreiben kann.

Der Einsatz dieses Moduls ist aber erst nach der Behandlung der Datenstrukturen sinnvoll, da die eingelesenen Daten in einer Liste gespeichert werden.

11.2 Literaturverzeichnis

Ernesti, Johannes, Kaiser, Peter: Python 3, Rheinwerk 2015

Kofler, Michael: Python Grundkurs, Rheinwerk 2020

Weigand, Michael: Python Ge-Packt, mitp 2017

11.3 Struktur der Datenbank

```
#daba00
import sqlite3
verbindung = sqlite3.connect("lkwMaut.db")
c=verbindung.cursor()
print('Datenbank angelegt!')
```

```
import sqlite3
verbindung = sqlite3.connect("lkwmaut.db")
c=verbindung.cursor()
c.execute("""CREATE TABLE IF NOT EXISTS lkw (lkwKz
TEXT(10),schadklasse TEXT (2))""")
for lkwdaten in (
            ("K-PY666","4A"),
            ("K-PY111","3B"),
            ("K-PY222","4C"),
            ("K-PY333","3D"),
            ("K-PY444","4F"),
            ("K-PY555","4E")):
    c.execute("INSERT INTO lkw VALUES (?,?)",lkwdaten)
verbindung.commit()
```

```
# Daba02
import sqlite3
verbindung = sqlite3.connect("lkwmaut.db")
c=verbindung.cursor()
c.execute("""CREATE TABLE gebuehr (schadklasse TEXT
(2),eurokm REAL)""")
for mautdaten in (("4A",0.187),
            ("3B",0.184),
            ("4C",0.208),
            ("3D",0.204),
            ("4F",0.261),
            ("4E",0.25)):
    c.execute("INSERT INTO gebuehr VALUES (?,?)",mautdaten)
verbindung.commit()
c=verbindung.cursor()
c.execute("SELECT * FROM gebuehr")

for zeile in c:
    a = list(zeile)
    print(a[0], a[1])
c.close
```

```
#daba04
import sqlite3
verbindung = sqlite3.connect("lkwmaut.db")
c=verbindung.cursor()
c.execute("""SELECT lkwkz, mautkm FROM fahrtenbuch WHERE
mautkm>300""")
for zeile in c:
    print(zeile)
c.close(
```

```
#daba04b_datum
import sqlite3
verbindung = sqlite3.connect("lkwmaut.db")
c=verbindung.cursor()
c.execute("""SELECT lkwkz, datum, mautkm FROM fahrtenbuch
WHERE datum ='02.10.2022' AND mautkm >300""")

for zeile in c:
    print(zeile)
c.close()


# daba05 achsenzahl =
import sqlite3

verbindung = sqlite3.connect("lkwmaut.db")
c = verbindung.cursor()
# --- Likek ----------------
c.execute("""SELECT lkwkz, schadklasse FROM lkw WHERE
schadklasse LIKE '4*' """)
for zeile in c:
    print(zeile)

print("--- Anzahl Einträge_____________________")
c.execute("SELECT lkwkz, count(*) AS anzahl FROM fahrtenbuch
GROUP BY lkwkz")
for zeile in c:
    print(zeile)
print(" Fahrtenbuch sortiert -----")
c.execute("SELECT * FROM fahrtenbuch ORDER BY lkwkz")
for zeile in c:
    print(zeile)
print("--- aus -----")
c.execute("""DROP TABLE zwidat3""")
c.execute("""CREATE TABLE IF NOT EXISTS zwidat3(lkwkz
TEXT(10), sumkm INTEGER)""")
print("-----Ausgabe zwidat3----------")

c.execute(
    """INSERT INTO zwidat3 (lkwkz, sumkm)
           SELECT lkwkz, sum(mautkm) FROM fahrtenbuch GROUP
BY lkwkz"""
)
c.execute("""SELECT * FROM zwidat3""")
for zeile in c:
    print(zeile)
c.execute(
    """SELECT lkw.lkwkz,zwidat3.sumkm * gebuehr.eurokm
             FROM zwidat3, lkw, gebuehr
             WHERE zwidat3.lkwkz = lkw.lkwkz and
lkw.schadklasse = gebuehr.schadklasse"""
```

```
)
print("gebuehren ----")
for zeile in c:
    print(zeile)

c.execute(
    """SELECT sum(zwidat3.sumkm*gebuehr.eurokm)
                    FROM zwidat3, lkw, gebuehr
                     WHERE zwidat3.lkwkz = lkw.lkwkz and
lkw.schadklasse = gebuehr.schadklasse"""
)
for zeile in c:
    print(zeile)

c.execute(
    """SELECT '15.10.2021' - datum
                FROM fahrtenbuch
                WHERE lkwkz = 'K-PY555'"""
)
for zeile in c:
    print(zeile)

c.close()
```

11.4 Sachwortverzeichnis